

**TRUSTED DATA PATH
PROTECTING SHARED DATA IN VIRTUALIZED DISTRIBUTED
SYSTEMS**

A Thesis
Presented to
The Academic Faculty

by

Jiantao Kong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computer Science
Georgia Institute of Technology
May 2010

Copyright © 2010 by Jiantao Kong

**TRUSTED DATA PATH
PROTECTING SHARED DATA IN VIRTUALIZED DISTRIBUTED
SYSTEMS**

Approved by:

Professor Mustaque Ahamad,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Douglas M. Blough
Electrical & Computer Engineering
Georgia Institute of Technology

Research Scientist Greg Eisenhauer
School of Computer Science
Georgia Institute of Technology

Professor Wenke Lee
School of Computer Science
Georgia Institute of Technology

Date Approved: January 9th, 2010

To my wife,

Ning Jiang,

for all of her love and support.

ACKNOWLEDGEMENTS

I would like to thank Dr. Karsten Schwan, for his brilliant guidance throughout my Ph.D study and also with my professional development. I would like to thank Dr. Mustaque Ahamad, Dr. Greg Eisenhauer, Dr. Wenke Lee, and Dr. Douglas M. Blough for thoughtful discussions and for serving on my committee.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	viii
LIST OF FIGURES	ix
LIST OF SYMBOLS OR ABBREVIATIONS	xi
ABSTRACT	xii
SUMMARY	xiii
 I INTRODUCTION	 1
1.1 Related Work	4
1.1.1 Access Control	4
1.1.2 Information Flow Control	7
1.1.3 Online Monitoring	9
1.2 Contributions and Limitations	11
1.2.1 Contributions	12
1.2.2 Limitations	14
 II THE CFC-TBAC MODEL	 17
2.1 Motivating Examples	17
2.1.1 Healthcare Information System	19
2.1.2 Cooperative Data Sharing	20
2.2 Context-Based Access Restriction	22
2.3 Trust-based Access Control	26
2.3.1 TBAC Model	26
2.3.2 TBAC Policy	28
2.3.3 TBAC Policy Engine	30
2.4 Context Flow Control	31

2.5	Others	33
III	CONTEXT EVALUATION	35
3.1	Context Evaluation Framework	35
3.1.1	TEvent Framework and Interface	36
3.1.2	TEvent Implementation	37
3.1.3	Communication Layer	39
3.1.4	Context Agent Communication	41
3.1.5	Integrity of Trust Agents	42
3.2	Basic Behavior Events	43
3.3	Context Agents	45
3.4	Context Agent Examples	47
3.4.1	Device Monitor	48
3.4.2	Guest Status	48
3.4.3	TCP Status	49
3.4.4	Session Tracking	49
3.4.5	Application Behavior	50
3.5	Experiments	50
IV	INTERCEPTION	53
4.1	Network Traffic Interception	53
4.1.1	Packet Interception	54
4.1.2	ProtectIT	55
4.1.3	Proxy-based Interception	57
4.1.4	Discussions	58
4.2	Secure Connection	59
4.2.1	SSL Introduction	59
4.2.2	Delegation of SSL Setup	60
4.2.3	SSL Repackaging	60
4.3	Interception Runtime	62

4.3.1	Runtime Interface	63
4.3.2	Runtime Composition	64
4.3.3	Data Exchange among Actionlets	67
4.3.4	Data Manipulation Actionlet	68
4.4	Experiments	69
4.4.1	ProtectIT Micro-benchmark	69
4.4.2	Proxy-based Interception Micro-benchmark	71
V	TRUSTED DATA PATH	77
5.1	TDP Software Components	77
5.1.1	Access Restrictions and Trust Vectors	77
5.1.2	Interception Rules and Runtimes	78
5.2	Deploy TDP Software	79
5.2.1	Deployment Interface	79
5.3	Applications	81
5.3.1	Care2X — Healthcare Information System	81
5.3.2	Online-storage and Email	82
5.3.3	Demonstration	83
VI	CONCLUSIONS AND FUTURE WORK	88
	REFERENCES	89

LIST OF TABLES

1	Restriction on Personal Identifiers	29
2	Translation of Trust-vector to Restriction	29
3	<i>TEvent</i> API	36
4	Basic System Events	44
5	Overhead of Context Agents and <i>TEvent</i>	50
6	Callbacks for Packet Interception Rule Registration	55
7	TCP Stack Simulator Interface	57
8	Overhead of Various Filters on an Intel C2D 2.6GHz Node	86

LIST OF FIGURES

1	Sample Health Care Information System	18
2	Example Data Propagation through Remote Storage	21
3	Context Evaluation by TBAC Engine	30
4	Flow Control on Trust-Data-Path Node	32
5	<i>TEvent</i> Framework	36
6	Basic Structure of <i>TEvent</i> Messages	37
7	Event Flow in <i>TEvent</i>	38
8	I/O descriptor ring of Xen (image copied from [2]).	40
9	Basic Structure of <i>Queries</i> and <i>Answers</i>	41
10	Flowchart of a Typical Context Agent.	45
11	Basic Template of Context Agent	47
12	CPU utilization of file SHA1 digesting	51
13	Transparent Proxy Based Interception.	58
14	SSL Handshake Example.	61
15	SSL Repackaging Demo	62
16	Interception Runtime Interface.	63
17	Runtime Composition	64
18	Base classes of Interception Runtime and Actionlet	65
19	Descriptor for Block Exchanged between Actionlets	67
20	Overhead of Intercepted Inward Traffic using ProtectIT	70
21	Overhead of Intercepted Outward Traffic using ProtectIT	70
22	Throughput of Intercepted Traffic using ProtectIT	71
23	Comparison of Latency on Receiving Data	72
24	Comparison of Latency on Sending Data	72
25	Throughput Comparison	73
26	CPU Utilization of Proxy-based Interception	74
27	Latency on traffic through HTTP runtime	75

28	Latency on SMTP Traffic	75
29	Latency on POP3 Traffic	76
30	Patient Information Shown on Different Reasonable Levels	84
31	Treatment Related Notes Shown on Medially Secured System	85
32	Restriction Filters on JPEG Images	87

LIST OF SYMBOLS OR ABBREVIATIONS

ABAC	Attribute Based Access Control.
ACL	Access Control List.
CFC	Context Flow Control.
DIFC	Decentralized Information Flow Control.
HIS	Healthcare Information System.
HTML	Hyper Text Markup Language.
HTTP	Hypertext Transfer Protocol.
IDS	Intrusion Detection System.
IFC	Information Flow Control.
MIME	Multipurpose Internet Mail Extensions.
PBA	Polymorphic Blending Attack.
POP3	Post Office Protocol Version 3.
RBAC	Role-based Access Control.
SMTP	Simple Mail Transfer Protocol.
SSL	Secure Sockets Layer.
SSN	Social Security Number.
TBAC	Trust-based Access Control.
TCP	Transmission Control Protocol.
TDP	Trusted Data Path.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
VM	Virtual Machine.
VMM	Virtual Machine Monitor.

ABSTRACT

When sharing data across multiple sites, service applications should not be trusted automatically. Services that are suspected of faulty, erroneous, or malicious behaviors, or that run on systems that may be compromised, should not be able to gain access to protected data or entrusted with the same data access rights as others. This thesis proposes a context flow model that controls the information flow in a distributed system. Each service application along with its surrounding context in a distributed system is treated as a controllable principal. This thesis defines a trust-based access control model that controls the information exchange between these principals. An online monitoring framework is used to evaluate the trustworthiness of the service applications and the underlining systems. An external communication interception runtime framework enforces trust-based access control transparently for the entire system.

SUMMARY

While sharing data across distributed machines is critical for modern IT applications, it also raises issues of maintaining desired data privacy and protecting data from inappropriate disclosure. However, it is difficult to retain controls on the data that is being shared in environments where services can be composed and deployed dynamically across distributed providers. To protect sensitive information against potential risks of inappropriate disclosures, access rights of applications to data should not only depend on their functional characteristics, but also on their as well as the underlying systems' behaviors. Stated more explicitly, applications that are suspected of faulty, erroneous, or malicious behaviors, or that run on systems that may be compromised, should not be able to gain access to protected data or entrusted with the same data access rights as others.

There exist many sophisticated prevention-based mechanisms to eliminate risks of inappropriate disclosures. However, there are cases where such risks are associated with the core functionality of the system. This thesis tries to provide a remedy for scenarios where such risks cannot be directly eliminated. The idea is to detect existing risks, then evaluate whether it is tolerable to share certain information under such risks.

This thesis proposes a context flow model (CFC) that controls the information flow in a distributed system. Each service application along with its surrounding context in the distributed system is treated as a controllable principal. CFC defines an access control model that controls the information exchange between these principals. The access control model has three main parts. First, an online monitoring framework is used to evaluate the trustworthiness of context of the service applications and the underlining systems. Second, a trust-based access control (TBAC) specification determines the permitted information

exchanges considering the active contexts of the service applications. Third, an external communication interception runtime framework enforces the above specification transparently for the entire distributed system. When there are multiple principals participating in the same information flow, the same TBAC specification is applied uniformly on all principals. In this way, we provide the protection guarantee throughout the entire information flow path, thus efficiently converting the path into a trusted data path (TDP).

The most important principle guiding the design and implementation of the CFC model is the integrity of the model itself. Since we do not trust the service applications and the underlying systems automatically, we place the risk evaluation and associated monitoring components of the CFC model into isolated domains, which are domains that are not subject to the same attacks or failures targeting applications and general purpose operating systems. We have implemented a prototype of trusted data paths leveraging virtualization technologies. The TDP software deploys online monitoring agents into privileged domains in platforms virtualized with the Xen hypervisor to assure the reliability of monitoring results. The TDP software also transparently intercepts communications between service applications, at the driver level in privileged domains. Using this technique, sensitive information that is not suitable for the current context can be automatically removed, without application involvement.

The TDP approach offers system support for protecting data access in environments where systems and services are subject to failures, programming errors, and attacks. It presents a system-level solution for fine-grained protection on data sharing in distributed systems. It particularly targets systems (1) that lack the extensibility to include context factors via built-in security mechanisms, such as legacy software; (2) that are subject to attack or are suspected of faulty behaviors themselves; (3) that wish to delegate context-based controls to external partners; and (4) that want to enforce context-based control ubiquitously instead of only at the source or sink. Applications that can benefit from the CFC-TBAC

model range from web applications like search and knowledge management or digital content services, to healthcare information systems, to file sharing systems using mail servers or online storage systems.

CHAPTER I

INTRODUCTION

Modern distributed systems are constructed as sets of cooperating service applications sharing data across distributed machines. Such service applications may be composed dynamically ‘on demand’, run locally or by external partners, and linked via dynamic or static contractual agreements. The spectrum ranges from traditional settings in which all service applications run in a single private data center to new approaches in which select service applications are dynamically offloaded to public clouds. In all such cases but particularly when services are running across multiple organizations, it is essential for applications to be able to safely share sensitive information. A typical healthcare information system [6] run by hospitals, for instance, shares patient information with doctors and nurses for treatment purposes, with external doctors for reviewing, with insurance agencies for billing, and with researchers for long term analysis, under the guidelines of federal and state regulations [63]. Operational information systems like those run by Delta Air Lines [47], interact with numerous external services, including for flight searches [42], ticket payment processing, revenue management, catering functions, and many others.

When sharing data across multiple sites, issues to be dealt with include maintaining desired data privacy, protecting the data being shared from inappropriate disclosures caused by unreliable or faulty partners [62]. However, modern systems make it difficult to retain the controls needed for maintaining strong data protection. For any given communication involving information sharing, there exist multiple hidden parties other than the data provider and the requester who have the opportunities to touch the data [56]. There are kernel modules that can ‘peek’ on the communication through socket buffers directly. Inside service applications, there are plugin extensions that can view the data even in plain

text mode. File writing operations and data output operations like those that send out emails from service applications, constitute additional ways to disclose data to new parties. Although it might be legitimate for these parties to access the data, they also represent potential risks concerning inappropriate data disclosures. Systems and applications' misbehaviors or programming errors can turn such risks into real damages. Attackers can exploit system vulnerabilities to create such parties, or compromise existing ones to steal the data.

This thesis seeks to develop improved system support for protecting data access in environments where applications rely on the use of shared or remote services, use shared underlying hardware, and where systems and applications are subject to failures, programming errors and attacks. The approach taken in this thesis and articulated in the following thesis statement is one in which applications operating on data are not trusted automatically.

Thesis Statement The access rights of applications to data should not only depend on functional application characteristics, but also on the dynamic application and underlying system behaviors. Stated more explicitly, services that are suspected of faulty, erroneous, or malicious behaviors, or that run on systems that may be compromised, should not be able to gain access to protected data and/or entrusted with the same data access rights as others.

There exist many sophisticated prevention-based mechanisms to eliminate some of the above risks, ranging from tools like general purpose anti-virus software [43, 46, 58], to corporate firewalls [7], to secure operating systems [39], to information system audit solutions and many others. However, there are cases where such risks are associated with the core functionality of the system. For instance, the risks of malicious plugins are a side effect of the service extensibility, the latter being an important property of service-based systems. This thesis tries to provide a remedy for scenarios where such risks cannot be directly eliminated. The idea is to detect existing risks, then evaluate whether it is tolerable to share certain information under such risks. Our research work is motivated by the

observations that different data has different levels of tolerance concerning data leakage and that different users have different risk profiles [48]. Concerning data, general personal information like email addresses, for instance, should be treated differently from financial information like credit card numbers. Concerning risk profiles, users within corporate firewalls operate with a different level of risk than those using public internet connection. As a result, it is desirable to permit controlled information sharing to achieve a balance between risks of inappropriate data disclosures and functionalities of application services.

This thesis proposes a context flow model (CFC) that controls the information flow in a distributed system. Each service application along with its surrounding context in the distributed system is treated as a controllable principal. CFC defines an access control model that controls the information exchange between these principals. The access control model has three main parts. First, an online monitoring framework is used to evaluate the trustworthiness of context of the service applications and the underlining systems. Second, a trust-based access control (TBAC) specification determines the permitted information exchanges considering the active contexts of the service applications. Third, an external communication interception runtime enforces the above specification transparently for the entire systems. When there are multiple principals participating in the same information flow, the same TBAC specification is applied uniformly on all principals. In this way, we provide the protection guarantee throughout the entire information flow path, thus efficiently converting the path into a trusted data path (TDP).

The most important principle guiding the design and implementation of the CFC model is the integrity of the model itself. Since we do not trust the service applications and the underlying systems automatically, we place the risk evaluation and associated monitoring components of the CFC model into isolated domains, which are domains that are not subject to the same attacks or failures targeting applications and general purpose operating systems. We have implemented a prototype of trusted data paths leveraging virtualization technologies. The TDP software deploys online monitoring agents into privileged domains

in platforms virtualized with the Xen hypervisor to assure the reliability of monitoring results. The TDP software also transparently intercepts communications between service applications, at the driver level in the privileged domains. Using this technique, sensitive information that is not suitable for the current context can be automatically removed, without application involvement.

1.1 Related Work

The TDP implementation of the CFC-TBAC model integrates multiple mechanisms to enforce data protection in potentially compromised environments. It is an access control mechanism in that it determines what kind of information is permitted to flow to a particular context. At the same time, it treats the multiple subjects that participate in the exchange of data as being part of the same data delivery path and enforces uniform flow controls on the information, thus presenting similarities with the information flow control model (IFC). The actual controls imposed are determined by dynamic risk assessments derived from evaluations of the service applications and their underlying systems, particularly considering properties that reveal whether there exists any party that is faulty or is subject to compromise. In this section, we explain our work in the context of access control, information flow control, and online monitoring.

1.1.1 Access Control

In computer security, access control is the basic mechanism that guards the ways in which data and resources are used. In any access control model, subjects represent the entities that perform actions, and objects represent the data and resources on which the subject operates. The access control procedure is to identify the subjects, and then grant permissions based on the access control matrix.

Traditionally, the subjects in an access control model link only to the users that control the software entities. Examples are the ID-based discretionary file access control in Unix systems, and role-based access control [15, 54] in many modern distributed systems. As

computer systems become more complex, such simple user-based controls may not be sufficiently rich to satisfy system security needs. For instance, in a peer to peer system, users are typically anonymous which makes identity-based authentication impossible. Environmental variables such as locations and times can also be important factors in determining the proper permissions on data and resources.

There is much prior research proposing access control models that incorporate various properties of the subjects into the access control matrix. Such models use these properties (attributes) of subjects either alone or together with the subjects as the ‘virtual identity’ to grant permissions, thus can be characterized as attribute-based access control (ABAC) models. The trust-based access control model in our context flow control system falls into this category in that it uses the trustworthiness properties of the service applications and the underling systems as the virtual identity to grant permissions. What makes our trust-based access control model unique is what properties we collect, how we collect such properties, how we organize these properties for easy policy specification, and how we enforce access controls based on such properties.

Factors that might affect access control decisions range from subjects’ past behaviors to environmental variables like location and time and many others. In open environments like peer to peer and pervasive systems, observations of an assessed agent’s behavior range from the satisfaction level of transactions [64, 61], to peer complaints [1], to contributions to the system [24], and to access history [22]. In various extended RBAC models, context extensions include temporal information [4, 28], and spatial information [9], and general environmental information [8] etc. On the one hand, our TBAC model presents the common features of all ABAC models in that it integrates both subject behaviors and environmental variables as the attributes of the subjects into the access control ‘model’. On the other hand, our model is distinct in that it focuses only on those behavior and environment ‘indicators’ that reveal abnormalities. The attributes we collect are measurements of differences between current vs. trusted application and system states.

The ways in which to collect state information can be placed into two categories. One uses direct observations of applications. The other uses state-collecting agents/sensors. The trust models of open systems are typical examples of the first category. [64] acquires the trust of one peer using observations about past transactions with this peer, and/or recommendations from others about this peer. It treats such recommendations carefully, by considering whether it has acquired satisfactory recommendations from recommenders in the past, and whether the recommendations match the ones from other trusted peers. The model in [61] uses direct/indirect trust and contributions. The evaluations of direct trust and contributions about one peer are from the observer's direct interaction history. The evaluations of indirect trust and contribution are calculated based on other observers' recommendations, multiplied by the direct trust about these recommenders. In [1], peer trustworthiness is assessed based on the number of complaints it files and the number of complaining peers. Here, a complaint is a report of cheating actions. Complaints are stored in a distributed fashion across all nodes, for scalability and robustness purposes. The integrity of stored complaints, however, is not apparent from the paper's description. [24] uses debit and credit to measure peer contributions, and derives peer-reputation accordingly. Reputation scores are stored locally, but managed by central servers and protected from tampering by public/private-key cryptography. [22] allows systems to evaluate requesters' trust based on past interaction logs and recommendations from peers in its confidential community. [8] represents another type of state collecting mechanism using context toolkits. It enforces authentications for components of the toolkits and ensures secure communications among them to form secure chains for reliably collecting state.

The trust state collection mechanism in our CFC-TBAC model falls into neither category, but is a hybrid of both. It observes application transactions and the system environment using trusted online-monitoring agents. What makes the mechanism unique is that these observations are not fed to the applications for access control. Instead, there are TBAC enforcement engines that perform permission granting procedures independently

from the data and resource providers. Access control is naturally considered as the data and resource provider's responsibility at the time of access, in both anonymous systems and identity-based systems, as demonstrated in the above mentioned systems. Our model separates this functionality from the applications running on behalf of the data and resource providers, since we do not trust such applications automatically.

A distinguishing attribute of the CFC-TBAC model is its fine-grained control over information sharing. Traditional access control procedures like those in the models discussed above either grant permissions for certain accesses or reject them. In contrast, the CFC-TBAC model can provide differentiated information sharing services to contexts with different trust properties. For instance, it can provide different versions, e.g. restricted views, of the same data to different users based on its assessment of the risks of improper data disclosures. The model also provides a systematical way to filter out sensitive information based on different dimensions of trust factors.

1.1.2 Information Flow Control

The CFC-TBAC model traces the flow of shared information in a distributed system. It uniformly applies the same TBAC specification on all principals participating in each flow. CFC-TBAC's goal concerning the control of information propagation is that same as that of the traditional information flow control (IFC) model, but there are several key differences, which are discussed in more detail below.

Information flow control (IFC) [5, 3, 10] is important to computer security. It enables controlling the release and propagation of protected information. The decentralized information flow control (DIFC) model in [45] extends existing IFC models by allowing individuals to declassify data they own, rather than requiring a central authority to do so. The DIFC model uses labels on slots that hold information – values. Each label contains an owner set and a reader sets for each owner to specify to which users owners are willing to release the information. Information flow is valid if it happens in a restricted way, e.g.,

by removing readers and/or adding owners. The correctness of the system depends on labeling slots based on well-defined security and integrity policies, and performing static and dynamic information flow checks based on slot labels. Implementation issues like those pertaining to the creation and location of the trusted code required for labeling, are not discussed.

Flume [37] provides a more practical solution for DIFC on stock operating systems. It implements DIFC at the granularity of processes, and integrates DIFC controls with standard communication abstract such as pipes, sockets, and files. Flume uses tags to describe the nature of the protected data, and uses a set of tags termed labels to describe what a privileged process has seen or can see. Unprivileged processes are not aware of the DIFC, thus cannot acquire any tagged information. Communications between privileged processes can happen only when the ‘belong to’ property holds for their labels. However, Flume still requires the distributed system to be implemented in a specific manner, e.g., splitting it into trusted processes and untrusted processes. It also requires a trusted operating system kernel.

DStar is another DIFC implementation, specifically targeting communications between processes running on different machines. It controls what a machine can send out based on what messages it receives. DStar is built on top of Flume [37] or HiStar [66]. It uses labels on processes and on messages to control communication. All labels form a partial order set of ‘can flow to’ relations. Trusted processes can acquire special privileges to bypass ‘can flow to’ restrictions. Untrusted processes have labels to express their security requirements, but have no privilege to omit any restriction.

Compared to the systems described above, the CFC-TBAC model targets the more general set of applications that are not aware of information flow controls and that contain both trusted and untrusted code in one domain. The model requires no application involvement in information flow control, since applications are labelled by trust vectors based on observations made by external monitoring agents and since enforcement is also performed

externally.

Although CFC-TBAC treats the application as a black-box, it can still trace the information flow into and out of the application based on patterns of use or on the metadata associated with inputs and outputs. As a result, it provides information flow control support to a much wider range of applications, but at the cost of not being able to cover all possible information flow paths.

1.1.3 Online Monitoring

The TDP implementation of the CFC-TBAC model relies on online monitoring agents to collect context states. The current implementation leverages existing monitoring mechanisms to evaluate the trustworthiness of the applications and the underlying systems, but we require that such mechanisms can evaluate contexts reliably and thoroughly. This is a non-trivial assumption for multiple reasons. First, while directly inspecting state via agents co-located with target applications and/or systems produces the most thorough results (since agents can easily access the states of the targets), agents' lack of isolation make it easy for malicious software to tamper with their inspection procedures and results. Conversely, while external network-based inspection offers excellent isolation, the information it can collect is limited to only low level network traffic, thereby constraining the range of attacks it can detect. New virtual machine-based inspection methods described in [21] provide good visibility for targets, while still providing strong isolation. The TDP implementation exploits such methods, i.e., VM-based monitoring, but also uses network based monitoring when indicated.

Regardless of which monitoring or inspection techniques are used, the TBAC model can take into account factors that include system integrity, the presence of malicious processes or network anomalies, unexpected application behaviors, and many others. TDP is implemented so as to leverages existing monitoring mechanisms to gather appropriate information and then inject it into the trust evaluation procedures used by the CFC-TBAC

model.

To check the integrity of a guest operating system, Garfinkel [21] proposes *virtual machine introspection* as an approach to intrusion detection. The IDS is co-located with the host but is isolated from it via the VMM. In [36], Kourai et al. present *Hyperspector* as an inter-VM monitoring mechanism. It provides software port mirroring, inter-VM disk mounting, and inter-VM process mapping to map server activities into the IDS VM for secure intrusion detection. Petroni [51] proposes *Copilot*, a coprocessor-based (using a PCI card) kernel memory monitor, to verify kernel runtime integrity through critical memory region hashing. The work is enhanced in [52] for specification-based semantic integrity checking.

An effective way to evaluate the security of a guest VM is to inspect what processes are actively running. Our prototype implements a naive agent to actively collect process creation and termination events in guest VMs, and reconstructs the process list outside the VM using these events to check whether there is software running for purposes that are good, such as anti-virus software, or are bad, such as rootkits and spyware. There exist more sophisticated mechanisms for VM-based detection of stealthy processes. In [26], Jones et al. present *Antfarm* to track the existence and activities of processes in guest VMs. It monitors the low level interactions between guest operating systems and the memory management in the VMM, such as page table operations related to virtual address spaces. In their later work [27], they identify hidden processes using statistical inference techniques by comparing trusted process information from *Antfarm* and untrusted process information from guest VMs. Jiang et al. [25] use *guest view casting* to externally reconstruct a semantically meaningful view of the guest's VM from its raw virtual memory, and then use comparisons between internal and external views to detect malware. Payne et al. present the XenAccess tool [49] with which a wide variety of guest VM inspection methods can be implemented.

Another way to detect system anomalies is through network traffic monitoring. In our

prototype, we install hooks at the network device driver level, to obtain all necessary information related to active network connections for analysis purpose. Simple methods like checking unexpected connections to known backdoors [57] can be easily implemented as a context agent. One way to enhance our anomaly detection ability is to employ more sophisticated network traffic analysis methods. For instance, in [60], Thottan et al. review several methods for anomaly detection in IP networks. In [23], Gu et al. propose to use *Maximum Entropy* to characterize the baseline distribution of network traffic. For payload-based detection, NETAD [40] and LERAD [41] model application payload data for intrusion detection. However, payload-based detection is often subject to polymorphic blending attacks (PBAs), as shown in [18]. In later work [17], Fogla et al. present a formal framework for PBAs and propose a technique to improve IDS against PBAs.

Evaluating specific applications' behaviors depends on their known behavior patterns. For simple patterns such as destination ranges of connections and folders for file saving etc., our prototype can directly verify them based on the collected system events related to network and file systems. For other patterns, Forrest et al. define *self* for privileged Unix processes, in terms of normal patterns of short sequences of system calls, in [19]. They show that it is simple and practical to detect several different classes of anomalies. Kalyan [30] et al. present a method for mail pattern analysis to scan outgoing emails from financial organizations to uncover violations of security policies. In [29], Jung et al. present *Privacy Oracle* to uncover applications' leaks of personal information in transmissions to remote servers. It uses a differential testing technique in which perturbations in the application inputs are mapped to perturbations in the application outputs to discover likely information leaks.

1.2 Contributions and Limitations

The trusted data path (TDP) approach offers system support for protecting data access in environments where systems and services are subject to failures, programming errors, and

attacks. It presents a system-level solution for fine-grained protection on data sharing in distributed systems. It particularly targets systems (1) that lack the extensibility to include context factors via built-in security mechanisms, such as legacy software; (2) that are subject to attack or are suspected of faulty behaviors themselves; (3) that wish to delegate context-based controls to external partners; and (4) that want to enforce context-based control ubiquitously instead of only at the source or sink. Applications that can benefit from the CFC-TBAC model range from web applications like search and knowledge management or digital content services, to healthcare information systems, to file sharing systems using mail servers or online storage systems.

The TDP implementation operates directly on the communications between service applications. Its independence from applications provides isolation guarantees to maintain data protection in potentially compromised environments, and it makes it suitable for wide range of applications. There are limitations, however, in terms of suitable target platforms (e.g., they must be virtualized), enforcement methods that rely on their ability to manipulate communications, the ability to properly trace information flows, and others.

1.2.1 Contributions

Leveraging the rapid evolution of virtualization technologies for modern hardware and software platforms [2, 38, 44], the trusted data path approach develops methods and software infrastructure to better control how data is accessed and used in distributed systems. The approach enhances underlying platforms to create and maintain *trusted data paths* for data shared across interacting services and machines. A rigorous context flow control model is used to control all communications, i.e., data sharing, among service applications on TDPs. The CFC-TBAC model, and the TDP implementation make the following contributions:

- *Trust-based, dynamic access control.* The CFC model adjusts data sharing in response to dynamically observed changes in service behaviors and/or in the properties of shared hardware or systems. Specifically, it enforces trust-based access control on

data flows to the services and platforms on TDPs. *Trust* is defined as the information provider's level of satisfaction about information safety, and it consists of scored ratings in multiple dimensions, i.e., trust vectors. This permits the CFC to base access control actions on factors such as whether the request is from an audited program, whether the system has anti-virus software running, and whether the request is from a reasonable environment like an office vs. a mobile machine. Further, there is flexibility in how trust values are determined from application behaviors and system contexts, and also in how trust values are mapped to access controls. Both are captured by policies associated with TDPs.

- *Trusted control agents isolated from applications.* TDP software places TBAC engines into privileged domains provided by the virtual machine monitor (VMM). This ensures that its checks and controls are not subject to the failures and attacks experienced by commodity operating systems and open applications. Our current implementation relies on the integrity of the VMM and its privileged domains.
- *External control.* Trusted data path implements message interception and manipulation transparently to and separately from the execution and implementation of applications running in guest VMs. This is done by intercepting the network traffic of guest VMs at the backend drivers in the driver domain, redirecting traffic to interception runtimes, and then reinserting processed results back into the normal data stream. This means that trusted data path can protect data exchanges between applications that do not have such functionality themselves, perhaps because they were not designed to do so (e.g., for legacy services) or because they were written to operate in some specific environment (e.g., within a single trust domain) but must now function in contexts where trust levels vary dynamically (e.g., when virtual machines migrate across physical platforms).
- *Flexible filtering.* The TBAC model offers permissions that are more flexible than

simple grant-or-reject actions. The permissions associated with the protected information are represented by restriction filters that filter out information not suitable for the recipients' contexts. The TBAC specification describes the relationship between different filters and various trust properties of the recipients, e.g., trust vectors, and the TBAC engine combines necessary filters based on the TBAC specifications to produce the restricted views suitable for the recipients' contexts.

- *End-to-end operation.* Access controls are associated with all participants of a trusted data exchange. Toward this end, TBAC engines running on different platforms cooperate with each other to enforce access control for all entities involved in a distributed data exchange. This is done by propagating access policy-related information together with the data being exchanged. As a result, access control enforcement is ubiquitous, extending across the entire data delivery path, hence the term *trusted data path*.

1.2.2 Limitations

The basic assumption, or rather requirement, of the TDP implementation is that there exist trusted places that can monitor service nodes and client nodes, and then perform trust-based access control operations. Our current implementation utilizes virtualization technologies for that purpose, by placing TDP functionality into the control domain (Domain 0) of the Xen hypervisor [2]. We believe this assumption to be acceptable given the increased use of virtualization in modern computer systems and the ever-increasing computational capabilities of future multicore platforms.

A key feature of trusted data paths is the ability to transparently and dynamically filter the data being shared. This requires that TBAC filters have knowledge about the semantics of ongoing communications. Moreover, this works only as long as such changes are acceptable to applications and to the communication protocols they use. This will not be the case for all application and protocols, of course. Additionally, we assume cooperative behaviors

from servers or clients if there is encryption involved in the communication protocol, such as the use of SSL/TLS [11]. When TDP encounters unexpected encrypted traffic that it cannot handle, it has to resort to the security policy to either ignore the traffic or downgrade the trust level of the parties involved.

TDP does not target applications like those in distributed finance or banking that depend on complete and correct information sharing. It also cannot be used to maintain strong data security, because (1) trust-based assessments may fail to recognize certain attacks or failures, so that data may be supplied to untrustworthy participants, (2) once data has been released, TDP controls cannot detect all of the ways in which data may be shared or acquired by malicious end users, and (3) TDP can only base access controls on the available context information, thus can only work as a supplement to existing access control mechanisms (e.g., role-based, capability-based, etc).

Compared to strict information flow control [10, 5, 3, 45, 66, 37, 67] offering systematic ways to identify an information flow using labels and tags, the CFC model lacks the mechanism to accurately trace each information flow. This is due to the way it operates, where the approach treats the application as a black-box. This means that it cannot match all of the inputs and outputs of an application due to reasons like data obfuscation. Since the applications we discuss here operate as black boxes, their outputs might be tainted by any of the inputs, which makes strict IFC not possible. Instead, we trace information flows based on the patterns of or the metadata associated with inputs and outputs. Finally, since we cannot monitor data manipulation and obfuscation inside the application, the CFC model cannot cover all possible information flows as in strict IFC.

* * *

The remainder of this thesis is organized as follows. In chapter 2, we give motivated examples and describe the CFC-TBAC model. In chapter 3, we then illustrate how TBAC engines monitor application and system behaviors using context agents. In chapter 4, we

introduce how we integrate TBAC into communication among applications of a distributed system. And finally we show how to apply the CFC-TBAC model on real systems, and how TDP software components are organized and deployed into distributed systems in chapter 5.

CHAPTER II

THE CFC-TBAC MODEL

The ability to share sensitive information is a key necessity for today's distributed applications. It is essential to provide access control to match security and privacy policies regarding such information. Existing solutions addressing the access control requirements typically require the data provider to evaluate the recipients' identities, roles, and/or capabilities owned, as well as their behaviors [1, 22, 24, 61, 64] and the surrounding context [4, 8, 9, 28] of applications running on behalf of the recipients. The context flow control (CFC) approach presented in this thesis adopts another way to deal with factors about application behaviors and surrounding context: 1) it moves the evaluation of such factors and the associated access control procedures into trusted places by leveraging virtualization technologies, 2) it works transparently and invisibly to the data provider and the recipient applications by enforcing additional context-based access restrictions directly on data objects in communication,

We will begin this chapter with two examples that demonstrate the usefulness of the CFC approach. Next, we explain that how we define the restrictions on basic data objects shared between applications in the distributed systems. We illustrate how we extend the access control using a trust-based access control (TBAC) model to cover the context information and how we evaluate those context information trustworthily. And last, we show how CFC constructs trusted data paths (TDPs) to offer end-to-end data protection.

2.1 Motivating Examples

This section describes two classes of applications able to benefit from the CFC approach and implementation. One is in the realm of healthcare information systems, and the other concerns cooperative data sharing, as illustrated in Figures 1 and 2. For these applications,

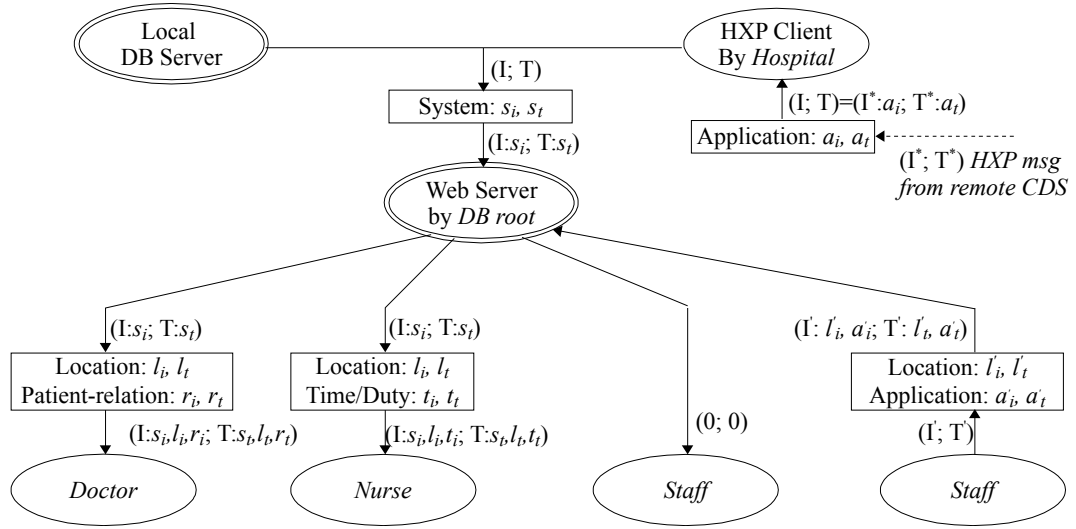


Figure 1: Health Care Information System.

trusted data paths can provide both (1) trust-based dynamic access control on information release, and (2) ubiquitous controls on information propagation, to match desired security and privacy policies. Both applications impose different demands on the sharing of sensitive information, thereby demonstrating the range of checks and controls supported by the CFC approach.

In the figures, ovals represent the services and client applications running on behalf of certain organizations or end users. These are the basic entities that own/generate, request/-consume, and propagate information. The double ovals represent the services that perform access controls on released information. The services and client applications rely on the underlying system for supportive functionalities, present certain behavior patterns, and execute on behalf of certain principals in variable contexts. All of the properties tightly related to information access are summarized as ‘context information’, illustrated in the figures as rectangles next to the corresponding services and applications.

The elements in parentheses — $(I:s_i; T:s_t)$ represent the information being exchanged between services and applications. Semicolons separate different fields of the information into different elements, such as $I:s_i$ for identification information and $T:s_t$ for treatment

information in a patient record. Uppercase characters represent the portion of the data originally granted to requesters by the existing access control model, and lowercase characters indicate the need to apply certain restrictions to the data before it is released to the recipient. These restrictions correspond to the context information shown in the rectangles.

Characterizing the communication channels between services and applications, their properties are listed in brackets as meta-information associated with each channel. Such information describes (1) the purpose of the communication channel, (2) certain properties of the entities linked by the channel, (3) the kind of data on the channel, etc. In general, meta-information concerns the properties of the information in a channel's messages, represented by M_{data} .

2.1.1 Healthcare Information System

With the increased use of electronic medical records, healthcare information systems (HISs) must address challenges concerning data privacy and security. Privacy requirements, for instance, are described in HIPAA [63] and in Directive 95/46/EC [14] in the U.S. and Europe. Since such regulations are formulated using natural languages and can be interpreted depending on the ever-changing environment, security procedures adopted in the healthcare information system of organizations like hospitals should be flexible to address emerging threats.

As a concrete example, consider the Care2X open-source healthcare information system [6]. Figure 1 illustrates a deployment of Care2X in a hospital. The core of the system is a web server working as the front-end interface. End users like doctors, nurses, staff and patients access the system using client side browsers. The Web server communicates with the backend database server to access patient records.

The Care2X system adopts a simple role-based access control model. Patients' healthcare information is categorized based the source of the information, such as nursing stations

and radiology, etc., and based on categories like laboratory test requests and results. Registered users acquire access rights for each category based their roles. However, the model does not address users' current contexts. For instance, we might expect doctors to get different views ($I:l_i$; $T:l_i$) of the same patient records when their access locations l change. One example is the case in which a doctor discusses the treatment of his patient in another doctor's office. For such a context, the system can place a strict restriction l_i on part of the patient identification information (I) and make only the related treatment information available.

There are other reasons than context for changing data access protection for patient records. Certain system or application behaviors may affect desired protection. For example, systems with uptodate security patches should have access privileges that are stronger than those that are not well-defended. Further, when an end user does not use the expected agent application to access the patient record system, this gives rise to concerns about the safety of the information released to and the integrity of the information received from this user. In other words, data access protection should change based on the current context a'_i and a'_i in which an end user operates.

2.1.2 Cooperative Data Sharing

A second class of examples driving our research concerns the extensive and cooperative sharing of data in today's highly networked society. Using Facebook and other social networking sites, people share data with their relatives, friends, and colleagues. The tacit assumption being made is that the data being shared flows only to those parties for which it is intended. Consider, for instance, someone updating his private photos in his online photo gallery to which he has granted access permission for his friends. A friend can access the photos on a private computer at home, or on public ones in say, an Internet Cafe or a library. In the second case, however, the machine's browser cache might leak the photos to unintended parties. Moreover, if the friend wants to share photos with other parties, there

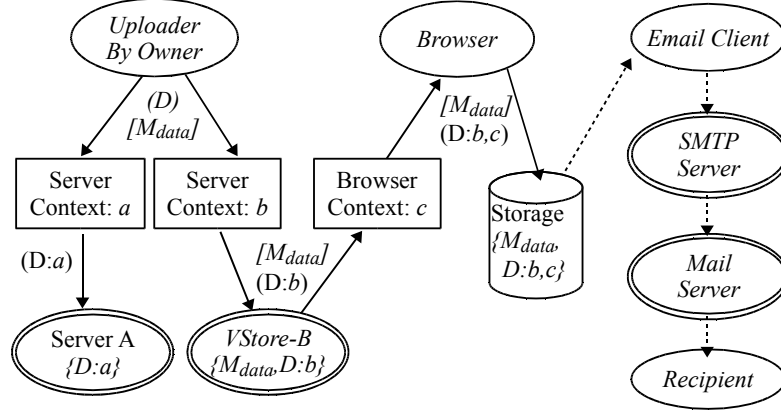


Figure 2: Network-based Data Sharing Example.

are few ways to prevent her from doing so except for the associated moral liability. Similar scenarios exist when sharing critical documents. For example, when an unpublished paper is sent to reviewers, the propagation of the paper should be accessed only by the small group of reviewers, except for its abstract and title.

As stated in the introduction, it is difficult to fully control information propagation in today's loosely coupled service networks. The purpose of our work is to strengthen such controls, on the basis of a common and trusted software platform. Such a platform should evaluate participating system and service behaviors, and it can then apply additional mechanisms to ubiquitously enforce data access policies throughout the information flow path. Information flows leaving the trusted set of platforms can be restricted in accordance with application-specific protection policies. The next paragraph further clarifies this.

In an ideal discrete access control model, a data owner can specify the desired access policy on information even after it has been released. Figure 2 represents such a policy as abstract meta-information M_{data} . In a server-based data sharing model, as shown in the left part of the figure, two servers may have different abilities to handle such meta-information. Well-equipped servers may strictly enforce the policy embedded in meta-information, as done in our previous work described in VStore [55] and even propagate the policy along with the data to the next node. A general-purpose server may not be able to do so. The

TDP approach to data protection describes such capabilities as ‘server contexts’, and the TDP software reacts to context differences by imposing the different restrictions a and b on the data seen by servers.

Continuing the example above, further along the path, the browser context c indicates whether information released to the browser is still under control. The context might include but is not limited to whether the browser understands M_{data} , whether M_{data} is still bound to the data when the browser sends it to storage or another email agent, whether the stored data is accessible by untrusted applications, etc. The paths from the data owner to the server, then to the browser, the storage, the email client, the mail server, and the email recipient constitute trusted data paths. The CFC model associates TBAC engines with each part of these paths to retain control over the data.

2.2 Context-Based Access Restriction

Consider a service-based distributed system offering a built-in security and privacy model that provides access controls on some object set \mathbf{D} . Such data objects typically contain ‘rich’ information for recipients, where the term ‘rich’ implies that the system can continue to function even if it only receives partial object information. For instance, in the Care2X HIS system, basic objects are the web pages sent to client browsers. Each such page consists of one or multiple elements, such as patient identification information, doctor notes, prescriptions, daily charts of blood pressures, and billing information, etc. Objects can be displayed and manipulated even when certain information is missing, such as a patient name not visible to nurses not on station. With the CFC, restrictions on data exchanges that hide critical information can depend not only on static identities, but also on the current roles of users, the times and locations of access, and similar contextual factors, as stated more precisely next.

The idea of the CFC and the TDP implementation realizing it is that information sharing should strike a balance between data safety and application functionalities, by placing

context-based access restrictions on data beyond those imposed by application-level controls. The CFC-TBAC model provides a systematic way of defining those restrictions for data recipients. More precisely, we use the CFC to determine a ‘restricted view’ of the object, thereby reducing its information content. The origin of a restricted view could be any member of an object group of the object set \mathbf{D} . This means that a restricted view is equivalent to a subset of the object set \mathbf{D} . Therefore, we can formally define an *access restriction* as a function $f : \mathbf{D} \rightarrow 2^{\mathbf{D}}$.

An access restriction f must satisfy two properties. First, it is obvious that an object must be one of the potential origins of its restricted view. Second, an access restriction should not create ambiguity. In other words, if a restricted view has a potential origin object d , then it must be equal to a restricted view of d . Formally, then, an access restriction can be defined as stated below.

Definition 1. For object set \mathbf{D} , *access restriction* is a function $f : \mathbf{D} \rightarrow 2^{\mathbf{D}}$, where $\forall d \in \mathbf{D}, d \in f(d)$, and $\forall \mathbf{D}' \in f(\mathbf{D})$: if $d \in \mathbf{D}'$, then $f(d) = \mathbf{D}'$.

we can easily prove that its function image $f(\mathbf{D})$ is a group of disjoint subsets of \mathbf{D} . If two subsets have at least one common element d , the restricted view of d should be equal to both subsets. Thus access restriction can also be described as follows.

Theorem 1. For object space \mathbf{D} , *access restriction* function f is equivalent to a group of subsets $\mathbf{D}_i \subseteq \mathbf{D}, 0 < i \leq n$, where $\cup_{0 < i \leq n} \mathbf{D}_i = \mathbf{D}$, and $\forall 0 < i < j \leq n : \mathbf{D}_i \cap \mathbf{D}_j = \emptyset$, and $\forall d \in \mathbf{D}_i : f(d) = \mathbf{D}_i$.

The formal model above captures the general principle of data reduction or filtering for objects. An implementation rule associated with such filtering assumed by the CFC model is that only those restricted views are permitted that are understood and accepted by the system and application. Stated intuitively, the view should be in the same form as the original object. Stated formally, $f(d)$ should be represented by an object d' in the object set \mathbf{D} or a slightly extended set \mathbf{D}^* (without loss generality, we use \mathbf{D} as the extended set

hereafter). This is equivalent to saying that some objects in \mathbf{D} are polymorphic. The system can treat such an object as either a single object or a potential object set if necessary. For instance, a blank field in a patient record means either there is no input or that the field is filtered out. So, when implemented in a system, the access restriction f can be in the form of a filter function $f^* : \mathbf{D} \rightarrow \mathbf{D}$.

One interesting point is that the same access restriction can be implemented in different filters depending on the selection of the representative objects. For example, for an object set $\{(a, b) | a, b \in \mathbf{N}\}$, the access restriction based on filters $f_1((a, b)) = (a + b, 0)$ and $f_2((a, b)) = (0, a + b)$ are identical. Both $(x, 0)$ and $(0, x)$ are representatives of the same set $\{(a, b) | a + b = x\}$.

Sometimes different context factors call for different restrictions on information release. This can be modeled as different access restriction filters applied to the same object set \mathbf{D} . Using a digital image as an example, it may be desirable to perform both resolution down-scaling and image greyscaling. The composition of the combined two restrictions filter out more information from the object than either one of them. To formalize the composability of access restrictions, we define a partial order \leq in definition 2 below. In plain text, the statement $g \leq f$ means g is less certain when tracing back to the original objects from the partial view. We define the composition of two access restrictions as their greatest lower bound, as in definition 3 below.

Definition 2. For two access restrictions f with image $\{\mathbf{D}'_i, 0 < i \leq m\}$ and g with image $\{\mathbf{D}''_j, 0 < j \leq n\}$, $g \leq f$ if and only if $\forall \mathbf{D}''_j, \exists \mathbf{I} \subseteq \{1, \dots, n\} : \mathbf{D}''_j = \cup_{i \in \mathbf{I}} \mathbf{D}'_i$.

Theorem 2. For two access restriction functions defined as $f, g : \mathbf{D} \rightarrow 2^{\mathbf{D}}$,

$$g \leq f \Leftrightarrow \forall d \in \mathbf{D} : f(d) \subseteq g(d)$$

Definition 3. For two access restrictions f and g in poset (\mathcal{F}, \leq) , the composition of f and

g is their greatest lower bound. Formally:

$$c = f \circ g \Leftrightarrow \begin{cases} c \leq f, c \leq g, \text{ and} \\ \forall k \leq f, k \leq g : k \leq c \end{cases}$$

For two access restrictions represented by filters $f, g : \mathbf{D} \rightarrow \mathbf{D}$, their composition is not always equivalent to the function composition of the two filters. This is because the filters use one object to represent a mapped subset of access restrictions, thus losing a certain degree of generality. For example, for access restrictions represented by filters ‘ $f(n) = 4 * (n/4)$ ’ and ‘ $g(n) = 5 * (n/5)$ ’, the access restriction composition is represented by ‘ $k(n) = 20 * (n/20)$ ’ and does not equal to either $g(f(n))$ or $f(g(n))$ obviously. Based on theorem 2, we have the relation of $f \circ g \leq g$, but not necessarily $f \circ g \leq f$. However, if the composition of filters f, g is commutable, we can easily prove that the filter function composition indeed represents the composition of restrictions as theorem 3.

Theorem 3. For two access restriction procedures $f, g : \mathbf{D} \rightarrow \mathbf{D}$ where $f(g(d)) = g(f(d))$, the composition of procedures $f \cdot g$ corresponds to access restriction composition, and it is represented by set of subsets of $2^{\mathbf{D}}$:

$$\begin{aligned} &\{\mathbf{D}_i \mid \mathbf{D}_i \subseteq \mathbf{D}; \quad \text{and} \\ &\forall d_1, d_2 \in \mathbf{D}_i : g(f(d_1)) = g(f(d_2)) \in \mathbf{D}_i; \quad \text{and} \\ &\forall d_1 \in \mathbf{D}_i, d_2 \in \bar{\mathbf{D}}_i : g(f(d_1)) \neq g(f(d_2))\} \end{aligned}$$

Specifically, if an object set \mathbf{D} has multiple dimensions, e.g., $\mathbf{D} = \mathbf{X} \times \mathbf{Y}$ where \times is the Cartesian product, then the composition of two filters f on \mathbf{X} and g on \mathbf{Y} is commutable and thus, it represents the composed restrictions on two dimensions.

$$\begin{aligned} f'(x, y) &= (f(x), y); \quad g'(x, y) = (x, g(y)); \\ f \circ g &:= f' \cdot g' \end{aligned}$$

There are some restriction filters that are not commutable, thus cannot guarantee the validity of the composition. For instances, consider the following two filters:

$$\begin{aligned} f(5m + i) &= 5m + 4, & \forall m \in \mathbb{N}, i \in \{0, 1, 2, 3, 4\} \\ g(4n + j) &= 4n, & \forall n \in \mathbb{N}, j \in \{0, 1, 2, 3\} \end{aligned}$$

For statement “ $g(x) = 8$ ”, we can deduct that the possible set of x is $\{8, 9, 10, 11\}$. If for “ $g \cdot f(x) = 8$ ”, then the possible set of x is only $\{8, 9\}$. However, for most of the procedures that are meaningful for the CFC, they are usually ones that remove certain subfields of objects. It is easy to verify that such procedures are usually idempotent and commutable when composed together. In this thesis, we focus only on filters that guarantee commutable compositions. There are some restriction filters that are not commutable. Our framework can include such filters, but correctness cannot be guaranteed formally when they are combined with other filters.

2.3 *Trust-based Access Control*

In the previous section, we describe the access restrictions that produce partial view of objects for different contexts. In this section, we introduce the trust-based access control (TBAC) model supplementing existing application-level access controls, by evaluating the surrounding trust-related context of a running process, and for all messages delivered to (or from) this process, applying proper access restrictions.

2.3.1 TBAC Model

The implementation of trusted data paths operates on communications among running processes. Thus, the objects on which it operates are messages on network connections. In the TBAC model, the Subjects to which the model grants permissions on messages are the surrounding Contexts of message recipients, thus the term Context Flow Control. We use access control lists (ACLs) instead of capabilities, and we try to categorize messages

based on the types of information they contain, and then link the ACLs based on the data types contained in messages. Therefore, an ACL entry determines the permissions on the associated data type granted to a certain context.

Considering the complexity of today’s computer systems, there exist many trust-related factors, including but not limited to access location, access time, device installed, and software behavior patterns, etc. It is unrealistic, therefore, to include all such factors in an ACL. Moreover, as computer system evolves, new threats occur, which might make an ACL too dynamic to manage. In response and enlightened by the role-based access control model (RBAC) [15, 16], we introduce the concept of ‘trust vector’ as the intermediate between contexts and permissions, hence the term trust-based access control (TBAC). The use of trust vectors limits the complexity of the ACL to the cardinality of the trust vector set.

The CFC defines *trust* as the information provider’s level of satisfaction about information safety. A trust vector is the evaluation of the surrounding context in various aspects represented in certain scales, and it consists of scored ratings in multiple dimensions. For example, a trust vector can be in the form of {“*Reasonableness of Access*”, “*System Security Level*”}. Accessing a patient’s record at home by a doctor might not be that reasonable, though it is still better than accessing it from a public library, for example. This permits the CFC to base access control actions on factors such as whether the request is from an audited program, whether the system has anti-virus software running, and whether the request is from a reasonable environment like an office vs. a mobile machine.

For each trust vector, the permissions on messages are in the form of access restrictions. Under the assumption that each dimension of a trust vector is independent, we associate restriction filters separately on each dimension. We then use function composition to form the final restriction filters that can produce the correct views of messages to the authorized recipients.

We next summarize the above description into the TBAC specification.

- *CTXS*, set of all possible active context.

$TVECS = TVEC_1 \times \dots \times TVEC_n$, set of possible trust vectors

$FLTS$, set of restriction filters.

$OBJS$, set of categories of message data.

- $trust_evaluation : CTXS \rightarrow TVECS$ — $(trust_evaluation_i : CTXS \rightarrow TVEC_i)$
the active context to trust vector mapping.
- $RESTRS = 2^{(FLTS \times OBJS)}$, the set of restrictions that are defined by restrictions on categories of message data.
- $RA \subseteq RESTRS \times TVECS$ — $(RA_i \subseteq RESTRS \times TVEC_i)$
a many-to-many mapping that indicate the restrictions associated with certain trust vector.
- $restriction_applied : TVEC \rightarrow RESTRS$, the mapping that determines the maximum restrictions applied on a trust vector. Formally:

$$restriction_applied_i(tvec_i) = \{ \bigcup r \mid r \in RESTRS, (r, tvec_i) \in RA_i \}$$

$$restriction_applied(tvec) = \bigcup_i restriction_applied_i(tvec_i).$$
- $permission : (OBJS \times RESTRS) \rightarrow FLTS$, the mapping that converts restrictions into real permission on one specific message category. Formally:

$$permission(obj, restr) = \circ d : d \in FLTS, (d, obj) \in restr.$$

2.3.2 TBAC Policy

Having specified the basic TBAC model, we now show how to specify desired data protection using the TBAC model. This specification starts from understanding what restrictions can be applied on the data objects. It associates the restrictions with the evaluation of potential risks of information misuse in the surrounding contexts, with risks represented as trust vectors. The specification formalizes the context evaluation from ‘totally untrusted’ to ‘fully trusted’ with a partial ordered trust vector set. It then links active contexts to trust vectors, as well as trust vectors to restriction filters.

Table 1: Restriction on Personal Identifiers

Field Group	Op Names	Filters
<i>Critical</i>	<i>none; full</i>	$(c = \text{remove_critical_id}); 1$
<i>General</i>	<i>none; full</i>	$(g = \text{remove_general_id}); 1$
<i>Hospitalization</i>	<i>none; full</i>	$(h = \text{remove_hosp_id}); 1$
<i>Identifier</i>	<i>none; hosp; gen; full</i>	$(c \cdot g \cdot h); (c \cdot g); c; 1$

Table 2: Translation of Trust-vector to Restriction

Reasonableness	Critical	General	Hospitalization	Identifier
<i>High</i>	<i>full</i>	<i>full</i>	<i>full</i>	<i>full</i>
<i>Medium</i>	<i>none</i>	<i>full</i>	<i>full</i>	<i>gen</i>
<i>Low</i>	<i>none</i>	<i>none</i>	<i>full</i>	<i>hosp</i>
<i>NOT</i>	<i>none</i>	<i>none</i>	<i>none</i>	<i>none</i>

The basic unit of data controlled by the TBAC model is the message exchanged between running processes. Restricted messages should be in the same format as the original messages. We assume that messages contain field groups that are independent, so that we can define restriction filters on each of them and then compose them. We implement restriction filters in ways that remove protected fields, so the composition of filters are commutable.

Table 1 shows the set of restriction filters that operate on the personal identifiers in the HTML page of the Care2X HIS. For one specific data field group, there are multiple available restrictions with distinct names. Each name is associated with a restriction filter (1 represents the identity function). We can define three field groups for the personal identifier, separately for critical data like social security number, general data like address, and hospitalization related data like the patient-id. Alternatively, we can define one group just as ‘Identifier’ and then define multiple named restrictions on it.

Table 2 defines the relation of trust vectors with the above restrictions for one specific trust aspect – reasonableness. The table lists all possible values for one specific dimension of the trust vector. Each value then links to the restriction filters in Table 1 by the field group name and the restriction name in the table. It shows that different field groups have different expectations on contexts. One example is that the critical personal data needs

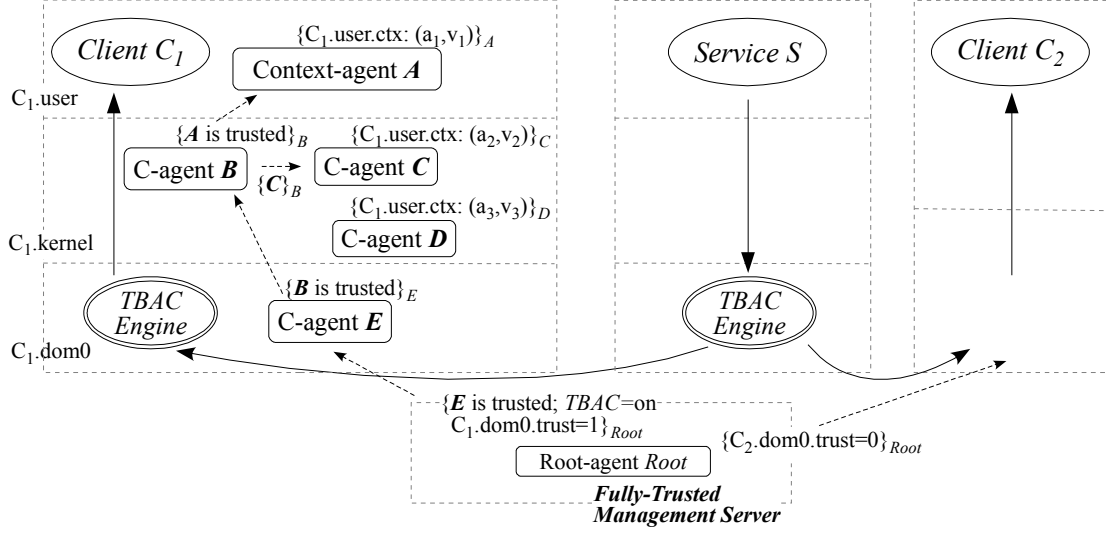


Figure 3: Context Evaluation by TBAC Engine.

the most protection. The mapping between restrictions and trust vectors should follow the natural meaning of trust vectors – less trust means more restrictions. A formal verification of such relations would be based on the partial order of the trust vectors and the restrictions.

The mapping of trust vectors to active contexts depends on how to translate the abstract description of information misuse risks to realistic system and application properties. In our model, such a mapping should be a formula that includes inputs from various online monitoring agents.

2.3.3 TBAC Policy Engine

The TBAC policy engine operates on application messages according to some predefined logic defined as policies. Since the correctness of the engine’s behavior depends on the trustworthiness of its surrounding context, it makes no sense to apply TBAC if we cannot guarantee the integrity of the input context information or the safety of the environment in which the engine runs.

Figure 3 illustrates how the TBAC policy engine collects context information and operates in a trusted fashion. The engine assumes that the lower layer of a system is more

trusted than its upper layer, as well as the existence of a fully trusted management server working as the root-of-trust. As seen in the figure, the client C_1 runs in a guest domain of a para-virtualized system. There are multiple context agents running in user space, kernel space, and in the control domain. They collect the systems' and applications' context information as inputs to the TBAC engine.

A context agent always signs its own statements. It can make statements about the trustworthiness of other context agents based on Rule 1. To assure the integrity of statements, the engine adopts transitivity rule 2 concerning the usage of the statement. As shown in Figure 3, statements from $\{A, B, C, E\}$ are valid for use by the TBAC engine, while the ones from D are not.

Rule 1. A context agent can make statements about its own layer and upper layers.

Rule 2. A context statement is valid to use in layer X if it is signed directly or transitively by a context agent from layer X or lower or by the root-of-trust.

TBAC engines run in contexts that are certified by the root-of-trust. In Figure 3, for a message from the service S to the client C_1 , the TBAC engine on the node S can acquire context statements about C_1 certified by the root-of-trust server, and performs the proper trust-based access control, or it can delay the check to the engine on the node C_1 . For messages to the client C_2 , the TBAC engine on S must perform the check with only statements about C_2 from the root, since there is no TBAC engine on C_2 .

2.4 Context Flow Control

We have specified how the TBAC engine collects context information and determines the current trust level and the associated restriction filters. We now describe the CFC model that uses TBAC to construct TDPs for existing service systems. The CFC model assumes that the service system consists of multiple communication channels that cooperatively deliver sensitive information from some source to some set of end users. These channels are typically socket connections between running processes on virtual machines.

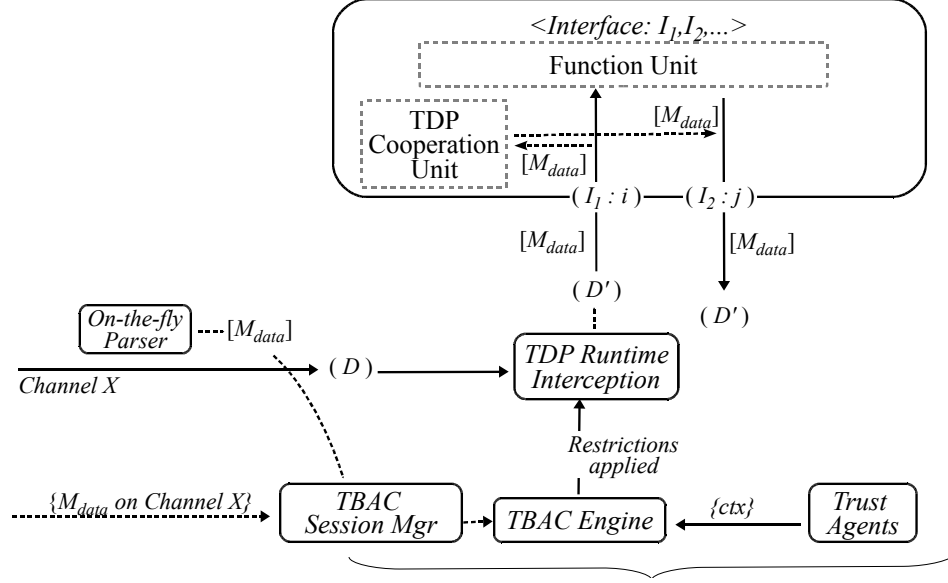


Figure 4: Flow Control on Trust-Data-Path Node.

The setup of TDPs starts by identifying the nodes that may serve as sources of sensitive information. From the source nodes, it traces the connections that carry the data to intermediate nodes and from intermediate nodes to end nodes. All these connections form a directed data delivery graph. The CFC model specifies interception rules to identify whether a connection belongs to the graph, what kind of data (e.g., the meta-information) is in the connection, and what TBAC specification applies. Thus, the entire data delivery graph is under control of the CFC, thereby forming a trusted data path.

Figure 4 shows a single service node that serves as both the sink and source of some sensitive information. We can view the node as a function unit with certain communication interfaces such as listening ports. A channel is automatically put under control of the TBAC session manager if it is part of a trusted data path identified by the interception rules. When a message comes in through the channel, the session manager first determines the meta-information M_{data} of the message. M_{data} is from either the cooperating applications or from the message itself, like customized HTML tags or MIME headers. The TBAC engine collects trust-related statements from the context agents and determines the current trust vector. It then calculates appropriate restrictions based on the policy specification

and acquires the final permission – a composition of restriction filters. Such filters are applied via on-the-fly message manipulation, using hooks at the device driver level. With such hooks, a message on a socket connection is redirected to a special process, termed the *interception runtime*. The TBAC engine applies the restrictions on data in that runtime, and it then pushes the result back into the connection. In this fashion, a restricted version of the message is delivered to the recipient, without its involvement and transparent to communicating parties.

The CFC model relies on properly maintained meta-information M_{data} to retain control over data, and M_{data} should always travel along with the sensitive data on TDPs. The CFC model supports propagation control by dynamically evaluating recipient nodes for whether or not they are able to maintain M_{data} of shared information concerning future propagation. Only for those with such abilities, the TBAC engine will green-light the information delivered to them. The trust vector in this case should include one or more dimensions about propagation control abilities. Examples of relevant context information include but are not limited to (1) whether the node has the ability to attach the meta-information M_{data} when relaying data to the next recipient, (2) whether the data is stored insecurely thus allowing access from uncontrolled applications, or (3) whether the node serves as a recipient-only without ‘save/print’ functions.

2.5 Others

The TBAC model delivers different versions of the same object to recipients with different surrounding contexts. The same technique has been used before for performance and protection purposes in various research works. In [59], Takagi et al. develop a system to transcode already-existing Web pages to be accessible using simplification or full-text transcoding modes. Knutsson et al. [32] show how server-directed transcoding can be integrated into the HTTP protocol and into the implementation of a proxy. In [65], Widener et al. present a mechanism for providing differential data protection to publish/subscribe

distributed systems using derived channels in ECho [13]. CameraCast project [33] implements logical device drivers for remote video sensors to provide differential views for users based on their capabilities.

There are services or client applications running on traditional non-virtualized platform. We have implemented protected data path [35] in linux based systems. It assumes that system kernels are safe and performs protection related work there. If the root-of-trust server can certify the trustworthiness of the kernel of a regular operating system, we can then trust the context information collected by agents running in the kernel, and even deploy the TBAC engine there to make it part of the trusted data path.

CHAPTER III

CONTEXT EVALUATION

The context flow control (CFC) approach intends to prevent inappropriate data sharing and access in distributed systems. It focuses on applications that are subject to failures, programming errors and attacks. Access rights of applications to data should not only depend on their functional characteristics, but also on their as well as the underlying systems' behaviors. The CFC-TBAC mode and the trusted data path (TDP) implementation adjust data sharing in response to dynamically observed changes in application behaviors and/or in the properties of underlying systems or hardware.

We design the context evaluation part to meet the following goals for supporting TDPs. First, the agents that collect context information should not be subject to compromised systems or faulty applications. Second, the model should be extensible to support new context agents to address emerging threats and new policies etc.

In this chapter, we introduce the light-weight event middleware that provides a unified interface to context agents and other parts of the TDP implementation. Next we describe how context agents collect systems and applications' behavior characteristics information, and how to guarantee the integrity of the collected information. Then we give examples of context agents in different categories.

3.1 Context Evaluation Framework

The TDP implementation consists of multiple functional components such as the context evaluation, session management, TBAC enforcement etc. Context evaluation is implemented by various standalone utilities, termed as context agents, that collect different systems and applications' behavior characteristics. We design a light-weight event delivery

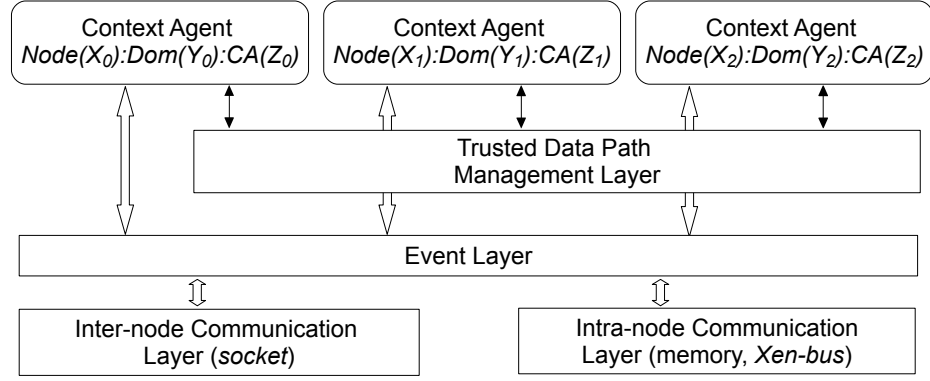


Figure 5: *TEvent* Framework

Table 3: *TEvent* API

API Name	Description
<i>event_register_member</i>	register unit as a member of <i>TEvent</i> .
<i>event_unregister_member</i>	register a member.
<i>event_invoke_member</i>	create an instance of specific member.
<i>event_release_member</i>	release an instance of specific member.
<i>event_send</i>	read events target to the specified member
<i>event_receive</i>	write events to targeted members

middleware to accommodate various TDP components into one framework. All components of the TDP implementation talk with each other through the unified event interface.

3.1.1 *TEvent* Framework and Interface

Figure 5 illustrates a light-weight event delivery middleware, termed as *TEvent*. The management layer crosses the entire distributed system deploying TDP components as configured. The deployment step is further discussed in chapter 5. Here we keep focus on context agents. Each context agent acquires a unique identifier recognized as the combination of node, domain, and index specifications. An event layer routes events (messages) among different agents based on the identifiers. The event layer utilizes either a socket based inter-node communication layer, or a shared memory based inter-VM communication layer to transmit events.

Table 3 shows the basic APIs for the *TEvent* middleware. In the TDP implementation,

```

struct tevent_message {
    int event_code;
    int dest_node, dest_domain, dest_index;
    int orig_node, orig_domain, orig_index;
    int event_flags;
    int event_size;
    union {
        // event_code-dependent structures.
    }u;
};

```

Figure 6: Basic Structure of *TEvent* Messages

functional components — usually standalone processes such as context agents, own distinct service names. Different service names represent different functionalities. A context agent calls *event_register_member* with its service name to register itself as a local member of the *TEvent*. Inside of the *TEvent* on each node, it maintains a table to map service names to internal identifiers shown in figure 5. Context agents talk with each other as well as other TDP components based on service names. To send events to another member of the *TEvent*, a context agent calls *event_invoke_member* to acquire the target’s internal identifier, and then sends/receives events by the identifier.

Figure 6 shows the structure of a *TEvent* event. The *event_code* field indicates the type of the event. The *dest_...* and *orig_...* specify the internal identifiers of the source and destination of a message. Various flag bits in *event_flags* indicate the requirements on message delivery and the expected responses from the target. A member can use *event_send* and *event_receive* to send and receive events in either blocking or non-blocking manners.

3.1.2 TEvent Implementation

Our prototype of the TDP implementation is based on the Xen [2] para-virtualization platform. Applications run in guest VMs. Context agents reside in either the same guest VMs as the applications, or in the control domain (Dom0) isolated from the guest VMs.

Figure 7 describes the detailed event flow in the *TEvent*. Context agents talk to the

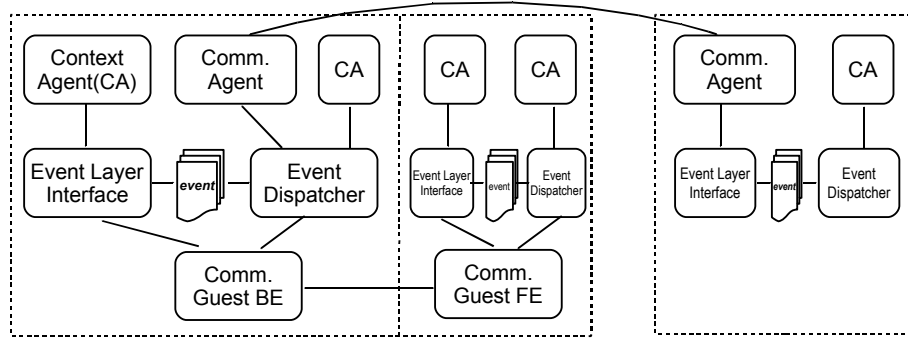


Figure 7: Event Flow in *TEvent*

event layer through a unified interface. The event layer maintains events from local context agents in a FIFO queue. An event dispatcher picks events from the queue and routes them based on destinations. Events that cross domains on the same node are exchanged between the guest backend and frontend. Events to remote nodes are routed to a communication agent in the control domain, then to communication agent peers on destination nodes.

The event layer interface and dispatcher are implemented as kernel modules, named as *TEvtCore*. The module maintains a table of active local members. Each entry of the table records one member's service name, internal identifier, owner process, and a small block of buffer for pending events. It also maintains hash tables for quick lookup based on service names or internal identifiers. The module registers a pseudo file system and creates one *inode* for each member, thus it can map the internal identifiers to file descriptors. The API calls of *event_send*, *event_receive* and *event_unregister_member* are then mapped to general system calls of *read*, *write* and *close*.

To send an event to a context agent, the caller need to lock up (e.g. increase reference count of) the target agent using *event_invoke_member*. For a local context agent, the *TEvtCore* does a hash table lookup by the service name. If there is no hit, the *TEvtCore* will contact the TDP management layer to deploy a function unit with the specified service name. The newly deployed unit registers itself so the lookup will success. Then the *TEvtCore* sends a control event *EVT_CTRL_INVOKE* to the agent on behalf of the caller. The

agent creates necessary data structures and claims that it is ready for processing incoming events from the caller. Optionally, the agent can spawn a child member of the *TEvent* for each caller, so the agent can handle events from different callers separately. Locking up a remote context agent works slightly different. Two *TEvtCore* modules talk with each other by control events `EVT_CTRL_INVOKE` and `EVT_CTRL_INVOKE_DONE` to lock up agents remotely. Unlocking a context agent (e.g. decreases its reference count) works similarly using the `EVT_CTRL_RELEASE` control event.

Events from *event_send* are put into a FIFO queue. An event dispatcher, which is a dedicated kernel thread, wakes up on new events and picks events from the head of the queue. If the destination of an event is a local context agent member, the dispatcher puts the event into the buffer specified by the member's table entry. The context agent reads in the event using *event_receive* call later. If the destination is another node or another VM on the same node, it puts the event into queues of the communication layer which is implemented as two special members described in the next section. The maximum size of the FIFO queue in our current prototype is four megabytes. Although it is more than enough for us to maintain events in the system without overflow, we implement a 'drop-the-oldest' policy and utilize time-out mechanisms to avoid infinite waiting on certain dropped events.

3.1.3 Communication Layer

For each *TEvtCore* in the control domain, there are two special members – a communication agent for inter-node communication and a communication guest backend for inter-VM communication, working as the communication layer of the *TEvent* middleware. The TDP management layer deploys the *TEvtCore* module, the communication agent, and the communication guest backend. Communication agents know each other through the management layer. Each communication agent owns its digital certificate. Any two communication agents can establish secure connections (SSL over TCP) using the certificates. A communication guest backend knows its counter-parts — communication guest frontends

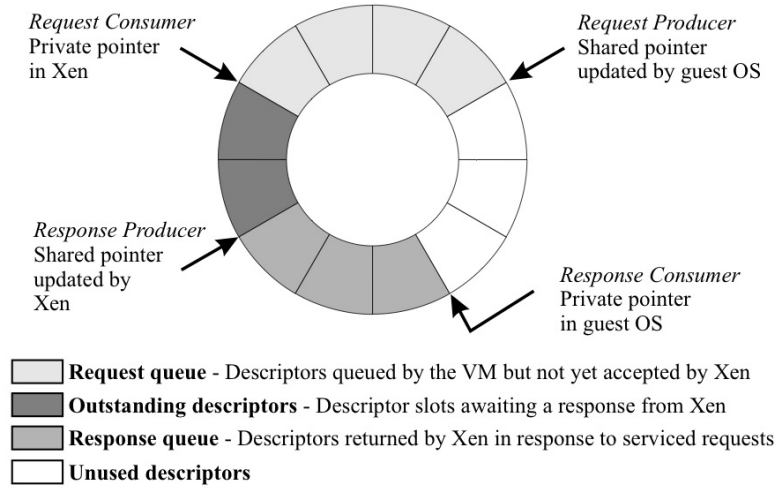


Figure 8: I/O descriptor ring of Xen (image copied from [2]).

in guest VMs by monitoring the creation of new VMs. They communicate through the shared-memory based Xen-Bus.

As a member of the *TEvtCore*, the communication agent has its own queue for pending events. The event dispatcher puts events for remote member in this queue. The agent reads in an event from the queue, checks the destination, and looks for the communication agent on the destination node. The event is then routed to the remote communication agent, and then to the destination through the remote *TEvtCore*.

The communication guest backend and frontend utilize Xen's I/O transfer mechanism to support event transfer between the Dom0 and guest VMs. We borrow a figure from [2] as shown in figure 8 to illustrate how Xen's I/O descriptor ring works. It is a circular queue of descriptors in a shared page between a guest VM and the Dom0. The communication guest frontend and backend are kernel modules in the guest VM and the Dom0 separately. The frontend allocates two ring buffers and grants access to the the guest backend. Access to each ring is based on two pairs of producer-consumer pointers. For events from a guest VM, the frontend places descriptors of events on a ring and advances the producer pointer. The backend removes the descriptors for handling and advances an associated consumer pointer. Acknowledgement of successful event handling are placed on the ring with another

pair of pointers. Events from Dom0 to the guest VM works similarly on another ring.

3.1.4 Context Agent Communication

Context agents communicate with each other in two ways. There is a pre-defined list of events known to all members of the *TEvent*. A context agent who is authorized to generate these events can broadcast these events to all agents in the Dom0 of the same node. Agents can define the range of events they are interested in to filter out unnecessary ones. They also use a pair of query and answer events for direct point-to-point communication. A agent embeds its query inside of a EVT_CTRL_QUERY event and sends it to the target agent. The recipient agent parses the query and embeds the answer into a EVT_CTRL_ANSWER response event.

```

#define FLAG_QUERY_MODE_ONCE           0x100000000
#define FLAG_QUERY_MODE_ONCHANGE      0x200000000
#define FLAG_QUERY_MODE_PERIODIC      0x400000000
#define FLAG_ANSWER_MOREPIECE         0x800000000
#define FLAG_ANSWER_MULTAPIECE_MASK   0x00000000F
struct tevent_query_body {
    int    index;
    int    flags;
    char   attribute[MAX_ATTR_LEN];
    char   parameter[MAX_PARAM_LEN];
};
struct tevent_answer_body {
    int    index;
    int    flags;
    int    seqno;
    char   answer[MAX_ANSWER_LEN];
};

```

Figure 9: Basic Structure of *Queries* and *Answers*

A context agent makes claim statements on context information using sets of attribute-value pairs. Each attribute has a character string name. The attribute name and the agent's service name together determines what meaning of the value is regarding to the current context. To query for specific context information, a caller first identifies which context

agent provides such information. It then invokes the agent to lock it up for the internal identifier. Next, it composes a query event with the desired attribute name and the associated parameter as illustrated in figure 9. A query event body also contains a unique index number to match to-be-received answers. Each query can be in one of three mode — answer once, answer when value changes, or answer periodically as specified by the query flag bits.

The corresponding answer event contains the same index to match the query. For the *onchange* and *periodic* modes, there are multiple answers for one query. It uses *seqno* in the answer event to provide a sequential indexes for those answers. In rare cases when an answer is too long to fit into one single answer event, it can set the `FLAG_ANSWER_MOREPIECE` bit in *flags* to indicate there are more pieces of events for the same answer. The order of multiple events of the same answer is identified by the low four bits of the *flags* as indicated by `FLAG_ANSWER_MULTIPIECE_MASK`.

3.1.5 Integrity of Trust Agents

On a virtualized platform, there are multiple places available to run context agents to monitor systems' and applications' behaviors. Running context agents as standard applications in guest VMs requires least privileges. The agents depend totally on system calls to collect information, thus have limited abilities on what information they can collect. Furthermore, they are subject to compromised system calls. Running context agents as kernel modules of guest operating systems brings the benefit of accessing critical system data structures directly. With proper instrumentation, an agent knows when and how an application interacts with the system and what resources it uses. However, it cannot guarantee the integrity and the completeness of the collected information by itself. Running context agents in a privileged domain provides the best isolation enforced by the VMM. However, the ability to access guest VMs is limited. Tools like XenAccess [49] are able to monitor guest VMs' memory and disk operations, but there are certain active information such as process

creation and termination that cannot be captured by passive monitoring mechanisms.

In our prototype, we implement hybrid deployment of context agents and utilize the chain of trust rule in section 2.3.3. Context agents themselves are certified before it can be deployed into TDPs. With the assumption of a trusted Dom0, we then assure that information from context agents in the Dom0 is reliable to use. For the purpose of collecting some systems' and applications' behavior information, we deploy a context agent in each guest VM as a kernel module. It acts on basic system behaviors such as process creation and termination, file operations, socket connection establishment etc. The integrity of the collected information have to be certified from outside of the guest domain since it is subject to compromised guest systems.

Threats to the event integrity come from two ways. First, events from the guest VMs can be forged by fake agents. Second, true events might be intercepted and falsified, or just dropped. In our prototype, events are associated with senders' identifications. The communication layer verifies whether the events are from the claimed sender processes. Agents in the Dom0 further verify the integrity of the communication layer and guest VMs. For instance, some events leave memory traces such as the 'task' for process creation in the guest kernel. Leveraging memory inspection tools such as XenAccess [49], we can check whether the events matches the current process list. Tools like [52, 51] can verify the integrity of the guest kernel thus guarantee that the agent and the communication layer is not compromised. Lares [50] can protect important kernel hooks from circumvention thus guarantee the events are properly delivered.

3.2 Basic Behavior Events

Context agents monitor systems' and applications' behavior based on interactions between applications and systems, and communications to and from storages and networks. Two types of information are useful for our model. First, we want to know what resources an

Table 4: Basic System Events

EVT_SYS_APP_EXEC EVT_SYS_APP_FORK EVT_SYS_APP_EXIT	<i>Events at process creation, fork, and termination time.</i>
EVT_SYS_TCP_LISTEN EVT_SYS_TCP_ACCEPT EVT_SYS_TCP_CONNECT EVT_SYS_TCP_RELEASE	<i>Events when applications listen on certain ports, accept connection requests, establish outbound connections, release connections.</i>
EVT_SYS_FILE_OPEN EVT_SYS_FILE_CLOSE EVT_SYS_FILE_READ EVT_SYS_FILE_WRITE	<i>Events when applications open, close files, and when their read or write pattern change.</i>
EVT_SYS_PKT_TCP EVT_SYS_PKT_UDP	<i>Events for incoming and outgoing network packets.</i>

application uses. Second, we want to act on certain application operations. Collecting information of the first type requires only simple queries on system states. For the second type, Linux, as the base of our prototype, is not equipped with enough hooks that we can learn of the events that we are interested in. In order to collect events related to processes, file operations, and socket operations, we implement a hook-based event notification mechanism in Linux. It consists of a list of hook functions, pieces of code to invoke those hooks in places where interesting events happen, and an API that allows kernel modules to install callbacks.

Table 4 lists the system events we collect to analysis systems' and applications' behaviors. There is one agent identified as '*Linux-Guest-Events*' responsible for the first three types, and another one identified as '*Guest-Packet-Sniffer*' for the last type. The first agent runs as a kernel module inside a Linux guest system. The second agent is a kernel module in Dom0 based on *ProtectIT* TCP interception described in [34]. The two agents generate events broadcasting through the *TEvent*.

Process-type events are useful to construct the active process list, detect unknown processes, check the existence of important software such as antivirus software. File-type

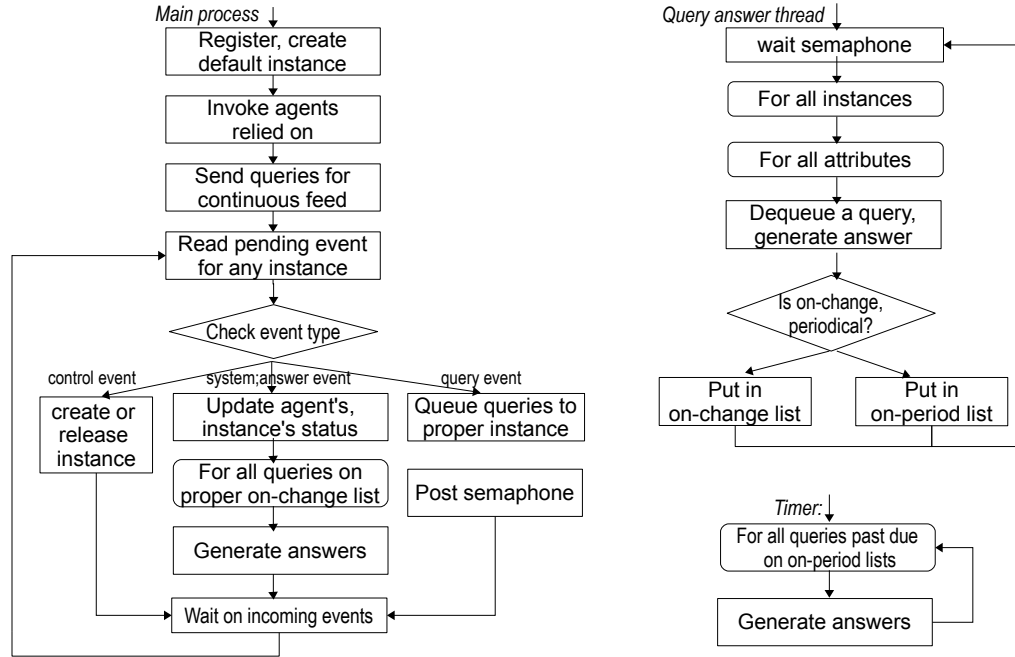


Figure 10: Flowchart of a Typical Context Agent.

events describe the working set of an application, and its access pattern—sequential access or random access. Socket-type and packet-type events can be used to detect suspicious connections and derive statistic traffic patterns for anomaly detection etc. There are other events left out of our prototype at this moment that are also useful. For instance, block level storage access can be an important supplement source of file access related events [49].

3.3 Context Agents

Context agents are certified entities that are trusted to provide reliable context information for TBAC engines. Except for low-level system event collection agents, most context agents are standalone processes running in the dom0 in our prototype. To collect context information, an agent accepts relevant system events, queries for events, and/or uses monitoring data produced by third parties. It then responds to queries from TDPs making claims about the current context in answer events.

Figure 10 describes the basic flow of a typical context agent. An agent first registers

itself to the *TEvent* and creates a default instance. It specifies its interests in basic system events and invokes necessary context agents which it relies on. Then it sends queries of either on-change mode or periodic mode to these agents for continuous event feeds. Next, it reads in pending events for all active instances. For control events, it creates new instances or releases existing instances. For basic system events and answer events from other agents, it updates the agent's and instances' status, generates answer events for active queries that are on-change based. The steps to acquire information from third party software is not listed in the flowchart as they are in a similar fashion. Finally, if the event is a fresh query, it attaches the query to the proper instance waiting for further processing.

There is a dedicated thread to process all queries. For each query attached on an instance, it generates an answer event first. After that, based on the access mode, it either discards the query, or puts it into an on-change query list or an on-period query list. There is a signal based timer to trigger new answer events for active queries that are in periodic mode.

As shown in figure 10, context agents share common structures. It is possible to provide a template that represents the common structure so programmers can focus on agent-specific details only. Figure 11 outlines the important pieces of an agent template in C++. For clarity purpose, parameters and return types are omitted for method signatures. Programmers derive classes from *TContextAgent*, *TAgentInstance*, *TContextAttr*, and *TContextQuery* to fill in agent-specific details. The derived classes need to implement virtual methods that begin with lower case letters. Methods begin with uppercase letters represent common steps usable by all agents.

Methods in *TContextAgent* represent the basic flow of an agent process. The derived class implements *newInstance* to create proper agent instances. The derived instance class of *TAgentInstance* handles system events and answer events from other agents in methods *onSysEvent* and *onAnswerEvent*, and updates the agent's and per instance's agent-specific

```

class TContextAgent{
    SetOption();        // set agent parameters for interests, registration etc.
    Register();         // register as TEvent member.
    MakeQuery();        // invoke agent relied on, send queries.
    ProcessEvent();     // event processing loop.
    newInstance();     // create new agent instance.
};
class TAgentInstance{
    AddAttribute();     // setup attribute-value pair.
    onSysEvent();       // process basic system events.
    onAnswerEvent();    // process answer events for previous queries.
    OnCtrlEvent();      // invoke, release instances.
    OnQuery();          // attach queries to proper attribute class.
    ChangeNotify();     // indicate certain attribute value changes.
};
class TContextAttr{
    OnChange();         // process attached on-change based queries.
    OnPeriod();         // process attached on-period based queries.
};
class TContextQuery{
    answerQuery();      // generate and push answer event out.
    isChanged();        // will new answer be different from the last one?
};

```

Figure 11: Basic Template of Context Agent

status. The class *TContextAttr* is the abstract of context claims. It maintains lists of active on-change and on-period queries represented by class *TContextQuery*. The derived query class of *TContextQuery* should implement *answerQuery* to create answer events and *isChanged* to tell whether the answer is different now from the last check.

3.4 Context Agent Examples

The TBD prototype uses context agents to determine three trust factors for our driver applications. First, data access operations should be reasonable in terms of locations of access, access-time, end user roles, etc. Second, data access requests should be from well behaved applications and systems. Third, the recipients of data should not inappropriately propagate data. The corresponding context agents that are able to collect such information are

described next. We introduce several typical context agents for evaluating context in these three aspects.

3.4.1 Device Monitor

Computer systems rely on attached devices to perform various functionalities. Network interface cards, either wired or wireless, provide communication support for network applications. In ubiquitous and pervasive computing, applications need all kinds of sensors to provide information about light, movement, proximity etc. In TDPs, we are concern about the impact on information safety of those devices. For instance, wired connections and wireless connections has different properties in terms of security. A location sensor can tell us whether a computer is in a place where it should be.

Ideally for detecting the location, we expect a sensor device like the Cricket sensor [53] which uses a combination of RF and ultrasound technologies to provide location information to attached host devices. Unfortunately, our prototype testbed does not have such devices. As an alternative solution, we assume that there are devices fixed in locations. So we can know of the computer location by checking the installed devices. In our experiment, we choose USB devices that have unique serial numbers. It can also use PCI based network interface cards with unique MAC addresses. The unique serial number is then mapped to the actual location by a pre-defined mapping table or a remote database query. Other queries supported by the agent include USB bus scan and PCI bus scan for device vendor IDs and product IDs. We can then answer device related questions such as whether the machine is using wired or wireless connections.

3.4.2 Guest Status

The security of a guest VM depends on many factors, the *guest status* agent provides an overview on what is going on inside of a guest VM. It monitors process creation and termination events to construct the list of active processes. It answers queries about whether a specific application such as anti-virus software is running, whether a specific connection

is from an audited program, whether an application is created locally or through a SSH session etc. Our prototype relies on system events described in section 3.2, and processes' pathname inside of the kernel data structure. Realistic evaluation should include more checks such as the authenticity of the executables.

3.4.3 TCP Status

Backdoors allow unauthenticated access to the system. Spyware collects information about users without their knowledge. Malicious plugins of an otherwise trustable application can leak user data to unauthorized parties. These network based attacks typically involve abnormal network traffics. In our prototype, we implement a simple *TCP status* agent to collect the TCP connection states and detect known anomaly. It can detect whether there are possible backdoors or spyware by checking listening ports and known bad connections, whether an application involves in suspicious data transfer by checking the number of established connections and the traffic volumes and destinations of connections.

3.4.4 Session Tracking

The CFC-TBAC model and the TDP implementation rely on properly maintained meta-information to retain control over data. To control the propagation of information along the entire data delivery path, we have to associate the access policy related meta-information with data on transfer. such meta-information can be attached as customized fields in communication protocols such as HTTP header fields or email MIME headers, or embedded inside of the data itself such as keywords etc.

A *session tracking agent* can keep records of meta-information and hashed digests of protected data objects to or from a node in order to trace the information flow. It either peeks on traffic along the TDPs by leveraging the transparent interception mechanisms in [34] and in chapter 4, or accepts inputs from cooperated applications and other components of TDPs. The records are indexed by data digests based on the SHA1 hash function [12], and the correctness relies on the rarity of collision of such hash functions.

Table 5: Overhead of Context Agents and *TEvent*

	Local (ms)	Remote (ms)
Context agent joining <i>TEvent</i>	0.0036	N/A
Invoking context agent (reuse instance)	0.0015	47.89
Invoking context agent (new instance)	0.0206	48.50
Process information (agent in Dom0 kernel)	0.0034	N/A
Context agent ability check	0.0143	0.41
USB device serial	5.12	7.03
Location by device serial	0.26	0.40
File digesting	1.69	N/A

3.4.5 Application Behavior

The *trust* in TDPs depends not only on the security of the system, but also on whether an application behaves as what it suppose to be. Application-specific agents can check whether an application behaves appropriately. For example, such agents might check rules concerning the servers with which an application can establish connections, the ports on which an application can listen for incoming requests, and/or where an application can store data, etc.

3.5 Experiments

In this section, we evaluate the performance of the event delivery system, and several typical context agents. The experiments are conducted in a giga-link LAN with multiple Xen platforms. The main testbed machine has an Intel Core2Duo processor of 2.6GHz with 4G memory, and a 2.1 GHz Core2Duo machine with 4G memory serves as the remote partner.

TDP software components use query events to collect context information from context agents. Here, we measure the basic latency for queries to those agents. The upper part of table 3.5 shows the one time cost to use the *TEvent* middleware setting up query channels to context agents. The lower part shows the latency of several typical query types: (1) an agent running in Dom0's kernel providing process information about guest systems; (2) a user-level agent answering queries about what kind of context it monitors; (3) a device status agent providing USB device serial numbers based on Vendor and Product IDs; (4)

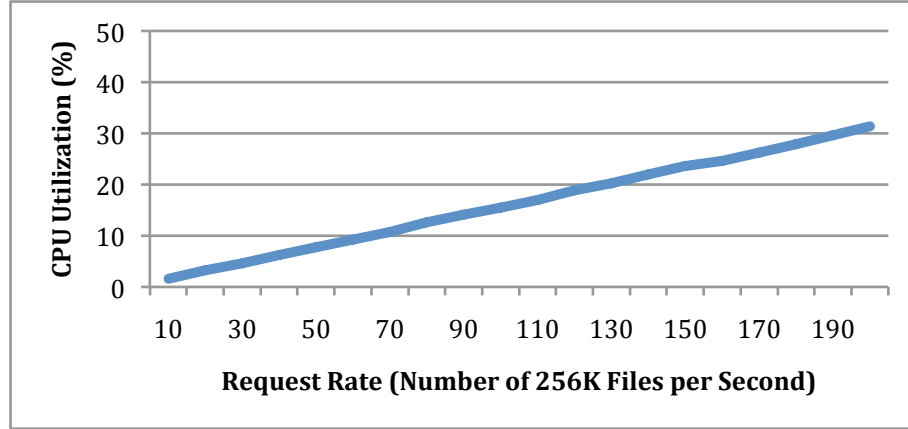


Figure 12: CPU utilization of file SHA1 digesting

an environment agent talking with a backend database to map device serial numbers to locations of the devices; and (5) a session tracking agent calculating the digest of a 256K size file. The table shows different overheads of various context agents. Compared to the latency of socket communication shown in later figures, the latency of context queries is mostly trivial. Moreover, some of the queries, such as for USB device serial numbers, incur only one time costs and can be amortized.

Context agents typically run passively and do not consume much computing resources. However, there are certain agents that actively monitor applications’ ongoing transactions. One example is the session tracking agent which keeps records of protected data objects that an application receives and sends. Figure 12 shows the CPU utilization when the agent calculates SHA1 hash [12] of an in memory file. Initially we send queries to the agent at various frequency and measure the CPU utilization using *getrusage*. However, *getrusage* does not return reliable values, neither does system utilities like *vmstat*. The reported utilization keeps lower than 2% until suddenly it jumps to more than 70%. So we use an alternative method that asks the agent to calculate digests for a file continuously (e.g. using 100% CPU time), and then derive the CPU utilization for various request frequencies. Results show that digesting requires moderate processing power – digesting 10 megabytes

data per second consumes around 6.2% of CPU time on a 2.6GHz Intel Core2Duo machine.

CHAPTER IV

INTERCEPTION

The CFC approach adjusts data sharing in response to dynamically observed changes in application behaviors and/or in the properties of shared hardware or systems. One of design goals is to control the information flow transparently and invisibly to communicating parties. To achieve this purpose, the TDP implementation performs packet interception and manipulation of network traffic of guest VMs at the driver level in the privileged driver domain, e.g. Dom0 in Xen. It redirects traffic to interception runtimes, and then inserts processed results back into the normal data stream.

In this chapter, we first introduce the driver level interception mechanism on a para-virtualized platform. We then introduce two methods of processing the network protocol layers of intercepted traffics. Next, we explain how to use interception runtimes to process application layer protocols including secure links, and how to integrate the TBAC enforcement to apply restriction based flow control on the protected data shared. Experiment results are presented at the end to demonstrate the effectiveness.

4.1 Network Traffic Interception

The Xen para-virtualization platform consists of a VMM (e.g. the hypervisor) that manages the hardware resources, guest VMs that run various operating systems, and a control domain (Dom0) that manages guest VMs and maps physical devices to virtual devices for guest VMs. The virtual network interface cards for a guest VM are linked to physical devices managed by the Dom0. A virtual network device driver in the guest system does not talk to hardware directly. Instead, it works as a front-end that relays device level packets between the guest system and the corresponding back-end in the Dom0. Using socket connections as examples, application data passes through network stacks to form IP packets,

a front-end driver passes IP packets to a corresponding back-end driver which then routes packets to the real NIC device.

There are multiple places where we can intercept a socket connection. One place is inside of the network stacks of the guest system kernel as described in our previous work [35]. It has the benefits of operating on application level payload data directly without worry about all the communication protocol details such as TCP/IP headers and packet order etc. However, the mechanism is subject to a compromised guest system. In the TDP implementation, we choose device driver level interception by leveraging Xen’s network I/O structures. The hypervisor provides the necessary isolation guarantee to assure the trustiness of the interception.

4.1.1 Packet Interception

The interception of network packets in the TDP implementation occurs at the network back-end driver level. We patch Xen’s network device management part with two hooks — when outgoing packets arrive from front-ends of guest systems and when incoming packets are ready to route to front-ends. A kernel module can register its callbacks if it is interested in communication between back-ends and front-ends. Each callback takes an IP packet as input and returns a boolean indicating whether the module consumes the packet or not. We also export two API functions that allow a module to inject new packets into either incoming or outgoing packet processing paths. So a module can consume the old packets of an intercepted socket connection, process the data in the packets, reassemble new packets with processed results, and inject them back to the original packet flow.

In our prototype, the core of the packet interception is a rule-based management module named *iccore*. A rule defines the socket connections to intercept and operations to apply on. To maximize the flexibility, the rule registration takes a list of callback functions as parameters. Table 6 lists all the callbacks necessary for the registration. The *iccore* enumerates all registered rules when it sees the first SYN packet of a TCP/IP connection. For

Table 6: Callbacks for Packet Interception Rule Registration

Callback Name	Description
<i>verify</i>	verify whether a packet matches the rule.
<i>instance</i>	create an instance of a matched connection.
<i>packetin</i>	process an incoming packet.
<i>packetout</i>	process an outgoing packet.
<i>release</i>	release an instance.

each rule, it calls *verify* for rule matching. If matched, it then creates an instance using *instance*. The *icore* maintains a hash-table of ‘IPs:ports’ to matched rules, thus future packets belonging to this connection are routed to either *packetin* or *packetout* efficiently.

There are two types of rules depending on actions applied on intercepted packets. Passive rule based interception peeks on packet data and does not affect the traffic. Active rule based interception modifies the packets, consumes the packets, and/or injects new packets to the intercepted connections. The current prototype allows one active rule and multiple passive rules per intercepted connection.

4.1.2 ProtectIT

The interception point at the back-end driver level is between the data link layer and the network layer. Packets contain application data, transport layer information (TCP/UDP headers) and network layer information (IP headers) at this point. However, the restriction filters in the TBAC model operate on application data only. To bridge the packet interception and the TBAC enforcement, we need extract application level data from raw packets based on the network layer and transport layer information.

ProtectIT is a self-contained TCP/IP interception implementation built as two kernel modules — a TCP/IP stack simulator and an interception manager. *ProtectIT* works on the bidirectional data of an intercepted TCP/IP connection. For intercepted packets, the stack simulator reorders and reassembles them to extract application level data. The data is then ready for processing through the interception manager. The interception manager organizes

all intercepted connections and relays the data to special application services, termed as *interception runtimes*. Processed data is pushed from interception runtimes down to the interface manager, then through the stack simulator who disassembles the data into small IP packets and injects them back to the connection.

The interception manager provides an interface for registering interception runtime and for making *ProtectIT* rules. Each interception runtime owns a unique runtime name. A *ProtectIT* rule defines the connections and interception runtime association using pairs of a connection matching pattern and a runtime name. A connection matching pattern specifies the ranges of TCP/IP source addresses, destination addresses, source ports and destination ports. The interception manager translates a *ProtectIT* rule into an rule used by the *iccore* by defining the *verify* callback based on the pattern specification. Details on how a matched connection is processed are explained below.

When there is a new connection that matches a *ProtectIT* rule, the TCP/IP stack simulator creates an instance for it. Application level data from subsequent packets is accumulated in the instance through *packetin* and *packetout*. The interception manager moves data from the instance to buffers for interception runtimes using APIs *copy_from_...* listed in table 7. The associated interception runtime uses *read* and *write* system calls to read data from and write processed results to buffers. Then the interception manager moves data back to the instance using APIs *copy_to_...* The TCP/IP stack simulator then processes the results and assembles new IP packets. The stack simulator maintains necessary status information about the intercepted connection, particularly for sequence numbers, acknowledge numbers, and checksums to make sure the integrity of the TCP/IP connection not compromised.

Table 7: TCP Stack Simulator Interface

Callback Name	Description
<i>copy_from_runtime_in</i>	move original data from intercepted connection to runtime.
<i>copy_from_runtime_out</i>	
<i>copy_to_runtime_in</i>	move processed result from runtime to TCP/IP stack simulator.
<i>copy_to_runtime_out</i>	
<i>data_runtime_in</i>	whether there is data available for processing by associated runtime.
<i>data_runtime_out</i>	
<i>space_runtime_in</i>	whether there is space available for creating new packets base on TCP window size.
<i>space_runtime_out</i>	

4.1.3 Proxy-based Interception

The TDP implementation provides another implementation of TCP/IP interception using transparent proxies. It utilizes the existing TCP/IP network stacks and the Netfilter framework in the kernel of the Dom0. The idea is to manipulate packet headers of intercepted packets so that the system treats the packets as a part of socket connections owned by proxy processes in the Dom0.

Using proxy-based interception, the TDP implementation assigns one TCP port to each interception runtime. The interception runtime creates a socket and listens on the assigned port. Here we use an example to illustrate the interception steps. Suppose a client application running in a guest VM establishes a TCP/IP connection to a remote server. The interception mechanism will convert the connection into two separated ones — *A* and *B* transparently. Connection *A* is from the client application to the interception runtime, and connection *B* is from the interception runtime to the remote server. Figure 13 shows the actual packet flow. For packets originated from the client application, it changes the destination fields so packets appear to be for connection *A*. For packets originated from the remote server, it changes the destination fields so packets appear to be for connection *B*. Similarly, for packets from the interception runtime on connection *A*, it changes the source fields to be the remote server so the client application thinks the connection is still intact,

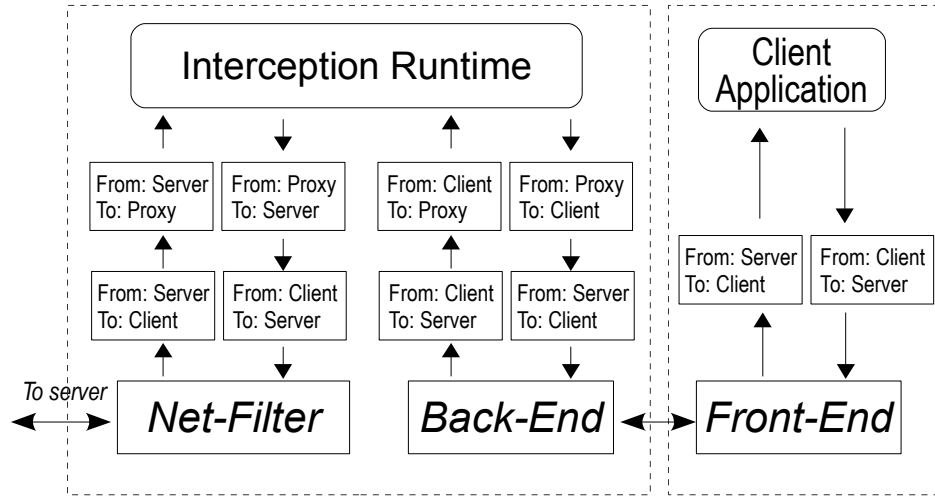


Figure 13: Transparent Proxy Based Interception.

e.g. from the remote server. The source fields of packets from the interception runtime on connection *B* are modified similarly.

The manipulation of packet headers for an intercepted traffic relies on the *iccore* module for packet to/from guest VMs, and on Netfilter for packets to/from remote nodes. Netfilter is a general packet-control framework within Linux. The packet reception and send routines inside of the Linux kernel invoke run through a list of registered hooks of Netfilter. There are different places for different types of hooks such as packet reception (PREROUTING), locally delivered (INPUT), forwarded (FORWARD), locally output (OUTPUT) and packet send (POSTROUTING). Our prototype use a PREROUTING hook for packets from remote nodes, and a POSTROUTING hook for packets to remote nodes.

4.1.4 Discussions

Both the ProtectIT and the transparent proxy based interception mechanisms have their advantages and limitations. The first mechanism has more control over the traffic. It can easily switch from interception mode to non-interception mode for the same connection, and it can easily peek at packets for monitoring purposes. With the second mechanism, interceptions are per connection, and this consumes local port resources, which may create

potential security issues. Further, monitoring the connection requires full interception. However, since the transparent proxy mode uses the existing system network stack, it can leverage that more stable and optimized code. Moreover, it is easy to set up the interception runtime in a dedicated domain, by manipulating only the packet headers.

Monitoring is another use case for passive interception. For instance, a network packet monitor agent can set up passive interception rules, peek on the intercepted packets without interfering the original packet delivery path. Another example is a context agent monitoring application level data of TCP/IP connections. It uses the ProtectIT mechanism with cloned packets. In this way, the agent works on duplicates and the connection keeps the original packets.

4.2 Secure Connection

Application level data extracted by the previously mention interception mechanisms contains both data objects and the necessary application level protocol information. It is straightforward to operate on data objects for a plaintext communication. However, when there is encryption involved in the protocol, handling of data objects requires knowledge about the encryption and decryption process. Secure Sockets Layer (SSL) [20] and the successor Transport Layer Security (TLS) [11], e.g. SSL 3.1 unofficially, are popular cryptographic protocols that provide security for communications over networks. In this section, we will explain how the TDP implementation addresses secure connections using SSL as an example.

4.2.1 SSL Introduction

The goal of the SSL protocol is to secure the communication between two applications. It uses TCP/IP on behalf of application-level protocols such as HTTP or SMTP. It allows the client and the server to confirm each other's identity, and to negotiate for an encrypted communication. A typical SSL session involves handshakes for server authentication and optional client authentication, and encrypted communication for deliver private application

data.

Server authentication in the SSL protocol uses digital certificates. A digital certificate contains the public key of the server, the validity period, domain names, and a digital signature of the certificate issuer. To authenticate a server, the client makes sure that 1) the certificate is in validity period, 2) the issuer is a trusted certificate authority, 3) the certificate is integrated based on issuer's signature, and 4) server's domain name matches. Once authenticated, the client uses the embedded public key to exchange secrets with the server. They then use the secrets to create symmetric keys for encrypted secure communication.

4.2.2 Delegation of SSL Setup

When a client application can be configured to use either SSL connection or plaintext connections, delegating SSL setup solves the problem easily. Using proxy-based interception as an example, the client-server connection is separated into one plaintext connection between the interception runtime and the client, and one secure connection between the interception runtime and the server. Since the interception runtime is in charge of the SSL connection, it operates on plaintext data with both connections. This solution can also benefit client applications that are not capable of secure communications. For example, we can use a simple *telnet* program to connect to a SSL-enabled mail server using this approach.

4.2.3 SSL Repackaging

When a client application need talk with the server directly using the SSL protocol, the SSL delegation method does not work anymore. The handling of the SSL protocol layers is inside of the server and the client application. The interception runtime can only get SSL records containing encrypted application data. The TDP software implements a SSL record decryption and re-encryption mechanism, termed as *SSL repackaging*, with help from the owner of the server's private key.

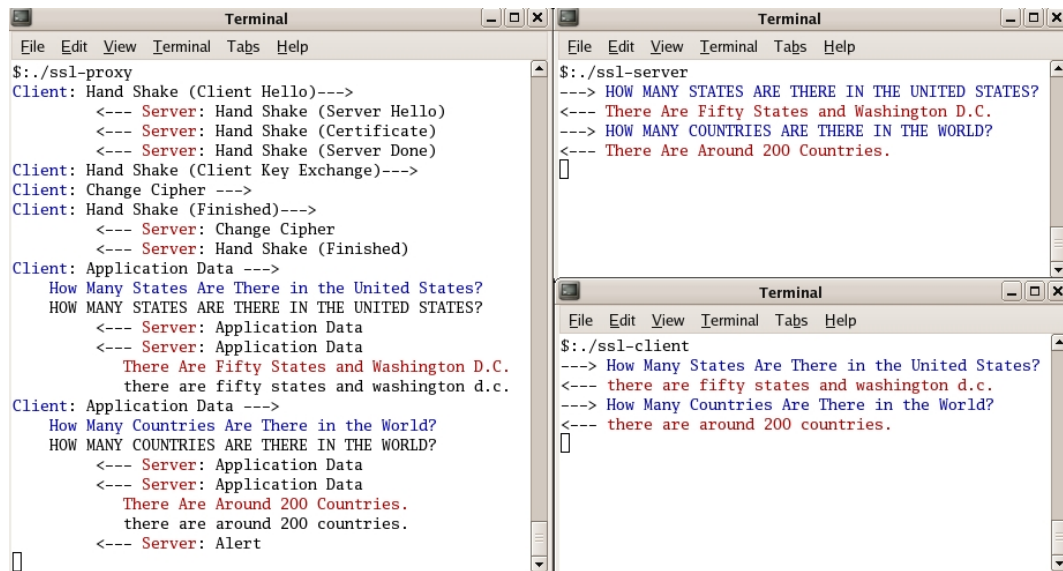
Figure 14 shows typical messages exchanged during a SSL handshake. The client sends a hello message including its SSL version number, cipher settings, compress settings and

Client		Server
Handshake:Client-Hello	→	
{ <i>version</i> ≤ 3.1; <i>random</i> ; <i>session_id</i> ; <i>cipher suites</i> ; <i>compress methods</i> }		
	←	Handshake:Server-Hello
		{ <i>version</i> =3.1; <i>random</i> ; <i>session_id</i> ; <i>cipher</i> = 'AES256_SHA'; <i>compress</i> =0 }
	←	Handshake:Server-Certificate
		{ <i>public key</i> ; <i>CA signature</i> ; <i>etc.</i> }
	←	Handshake:Server-Hello-Done
Handshake:Client-Key-Exchange	→	
{ <i>PreMasterSecret</i> } _{<i>server.publickey</i>}		
Handshake:Change-Cipher-Spec	→	
Handshake:Finished	→	
	←	Handshake:Change-Cipher-Spec
	←	Handshake:Finished
Application-Data	↔	Application-Data

Figure 14: SSL Handshake Example.

other information. The server replies a hello message with the selected SSL version, cipher algorithm, and compress method etc. The server then sends the client its certification for authentication. Once authenticated, the client creates the pre-master-secret and encrypts it with the server's public key. The server gets the encrypted pre-master-secret and decrypts it using its private key. Now both the client and the server know the pre-master-secret, so they can generate the same session key. After the client and the server issue the *Change-Cipher-Spec* message, following SSL records are then encrypted using the selected cipher algorithm with the session key.

The exchange of cipher suites and the choice is in plaintext, so we know what cipher algorithm that two communicating parties agree on. However, the pre-master-secret is encrypted with the server's public key and can only be decrypted using the corresponding private key. The TDP implementation assumes that it can get help from the private key owner — through some services running along with the server. With this assumption, the interception runtime acquires the decrypted pre-master-secret, generates the same session



```
Terminal
File Edit View Terminal Tabs Help
$ ./ssl-proxy
Client: Hand Shake (Client Hello)--->
<--- Server: Hand Shake (Server Hello)
<--- Server: Hand Shake (Certificate)
<--- Server: Hand Shake (Server Done)
Client: Hand Shake (Client Key Exchange)--->
Client: Change Cipher --->
Client: Hand Shake (Finished)--->
<--- Server: Change Cipher
<--- Server: Hand Shake (Finished)
Client: Application Data --->
How Many States Are There in the United States?
HOW MANY STATES ARE THERE IN THE UNITED STATES?
<--- Server: Application Data
<--- Server: Application Data
There Are Fifty States and Washington D.C.
there are fifty states and washington d.c.
Client: Application Data --->
How Many Countries Are There in the World?
HOW MANY COUNTRIES ARE THERE IN THE WORLD?
<--- Server: Application Data
<--- Server: Application Data
There Are Around 200 Countries.
there are around 200 countries.
<--- Server: Alert

Terminal
File Edit View Terminal Tabs Help
$ ./ssl-server
---> HOW MANY STATES ARE THERE IN THE UNITED STATES?
<--- There Are Fifty States and Washington D.C.
---> HOW MANY COUNTRIES ARE THERE IN THE WORLD?
<--- There Are Around 200 Countries.

Terminal
File Edit View Terminal Tabs Help
$ ./ssl-client
---> How Many States Are There in the United States?
<--- there are fifty states and washington d.c.
---> How Many Countries Are There in the World?
<--- there are around 200 countries.
```

Figure 15: SSL Repackaging Demo

key, and decrypts and re-encrypts the SSL traffic. The implementation of *SSL repackaging* leverages the openssl library so we can use its encryption, decryption and HMAC procedures.

Figure 15 demonstrates a successfully intercepted SSL connection. All SSL records after the ‘Change Cipher’ message are encrypted, including the ones containing questions from the client and the answers from the server. The proxy changes the text from the client to all uppercases, and changes the text from the server to all lowercases.

4.3 Interception Runtime

When a packet from an intercepted connection reaches an interception runtime, it contains both real data and application level protocol information such as HTTP or SMTP etc. The interception runtime unwraps the application protocol layer, extracts the real data, applies TBAC specified restriction filters, rewraps with application protocol information and feeds the final result to the original connection. We will describe the interface and the components of an interception runtime in details below.

4.3.1 Runtime Interface

An interception runtime is characterized by the types of application layer protocols and data objects that it handles. The TDP implementation assigns distinct names to different interception runtimes. An interception runtime registers itself by *runtime_register* with its unique name. It then calls *runtime_accept* for intercepted connections. The *runtime_accept* call returns four file descriptors — two read-only ones and two write-only ones representing the input and output points of an intercepted connection separately. The interception runtime uses two read-only descriptors to read in the inward and outward traffic data, and outputs processed results to the two write-only descriptors.

<i>runtime_register(name)</i> → <i>runtime</i>
<i>protectit_runtime_register(name)</i> → <i>runtime</i>
<i>proxy_runtime_getport()</i> → <i>port</i>
create socket <i>sock</i> and listen on <i>port</i>
<i>proxy_runtime_register(name, sock)</i> → <i>runtime_{proxy}</i>
combine <i>runtime_{proxy}</i> and <i>sock</i> → <i>runtime</i>
<i>runtime_accept(runtime)</i> → <i>rd_{inward}, wr_{inward}, rd_{outward}, wr_{outward}</i>
<i>protectit_runtime_accept(runtime)</i> → <i>fd_{inward}, fd_{outward}</i>
<i>fd_{inward}</i> → <i>rd_{inward}, wr_{inward}</i> ; <i>fd_{outward}</i> → <i>rd_{outward}, wr_{outward}</i>
accept new connection on <i>runtime</i> : <i>sock</i> → <i>conn_A</i>
<i>proxy_runtime_check_orig_dest(conn_A)</i> → <i>destination</i>
setup socket connection to <i>destination</i> → <i>conn_B</i>
if <i>conn_A</i> is outbound
<i>conn_A</i> → <i>rd_{outward}, wr_{inward}</i> ; <i>conn_B</i> → <i>rd_{inward}, wr_{outward}</i>
else
<i>conn_A</i> → <i>rd_{inward}, wr_{outward}</i> ; <i>conn_B</i> → <i>rd_{outward}, wr_{inward}</i>

Figure 16: Interception Runtime Interface.

TDP software implements two TCP/IP interception mechanisms in the kernel space of Dom0. The interception runtime framework works with both mechanisms. A thin interface layer translates the interception runtime APIs into ones used by the interception manager module of the ProtectIT and the proxy manager module of the transparent-proxy interception. Figure 16 shows the mapping from runtime APIs to APIs of the interception manager module and the proxy manager module.

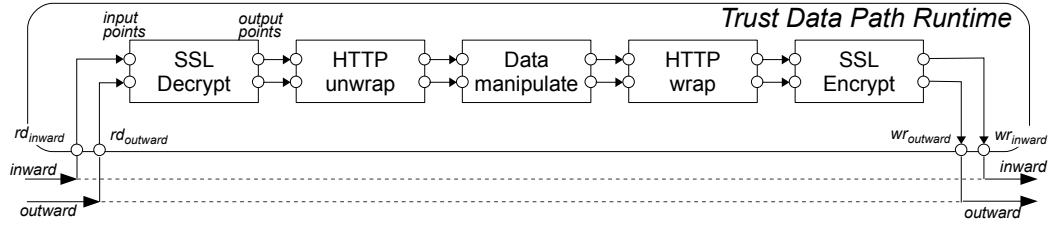


Figure 17: Runtime Composition.

For ProtectIT-based interception, the translation is pretty straightforward as shown in the figure. For transparent-proxy based interception, mapping for interception runtime APIs involves several steps. Each interception runtime needs a free port and creates a listening socket on it. The registration of the interception runtime takes both its name and the socket as input. Accepting a new intercepted connection involves two socket connections from the interception runtime to both ends of the intercepted connection — accepting a socket from one end and establishing one to another end. The interception runtime then sits between the two sockets working like a transparent proxy. The four file descriptors of the interception runtime are then mapped to the socket descriptors of two socket connections separately.

4.3.2 Runtime Composition

The interception runtime needs to handle both application layer protocols and data objects. Although our prototype does not limit how an interception runtime should be implemented, the workflow reveals the possibility of a modularized implementation. A runtime consists of series of function units that perform protocol unwrapping, data object manipulation, and protocol wrapping, respectively. By making each function unit an independent code piece, it enables the reuse of same units in different interception runtimes. For instance, interception runtimes that intercept HTTP traffics with different data objects can share the same HTTP protocol unwrapping and wrapping code.

The composition of runtimes depends on the desired workflow on intercepted connections. Figure 17 shows an interception runtime targeting secure HTTP connections. The interception runtime links together five function units that perform SSL decryption, HTTP unwrapping, data object manipulation, HTTP rewrapping, and SSL encryption in turn. Because each network communication contains both inward and outward traffic when viewed from one side, the function unit, termed *actionlet*, handles both traffic. The interception runtime feeds the data from the read-only descriptors rd_{inward} and $rd_{outward}$ to the two input points of the first actionlet. The actionlet processes the data and relays the processed result to the next actionlet. The last actionlet feeds the final result to the two write-only descriptors wr_{inward} and $wr_{outward}$.

```

class TActionletBlock; // block of data passed by actionlets
class TActionlet{
    TRuntime *runtime; // runtime instance it belongs to.
    TActionlet *prev, *next; // to form actionlet chain in runtime
    InputData(TActionletBlock*, int); // input points of inward/outward traffic
    TActionletBlock* OutputData(int); // output points of processed traffic
    int GetStatus(void); // whether it can do something
    int Resume(void); // process data and pass to next actionlet
};
class TRuntimeFlow{
    int fd_rd_in, fd_wr_in; // file descriptors for input from interception
    int fd_rd_out, fd_wr_out; // file descriptors for output to interception
    TActionlet *actHead; // head of the actionlet chain
    TGateActionlet *actTail; // tail of chain to interact with interface layer
    .....
    int InsertActionlet(TActionlet *); // setup actionlet chain
    int Resumable(void); // whether there is anything to do
    int Resume(void); // call all actionlets to do what they can
};

```

Figure 18: Base classes of Interception Runtime and Actionlet (*simplified*)

Interception Runtimes and actionlets are implemented as C++ classes or C structures with associated functions in our prototype. Figure 18 shows the simplified base class definition. *TRuntimeFlow* represents the workflow on an intercepted connection, and *TActionlet*

is the abstract of actionlets that form the workflow. Programmers derive their classes from these two to cover runtime-specific details. For illustration purpose, we use the name of the base classes for any derived classes hereafter. *TRuntimeFlow* contains four file descriptor to interact with in-kernel interception. There are two derived classes *TActionletEntry* and *TActionletExit* working as default end points for the workflow interacting with the file descriptors.

For a concrete interception workflow, an instance of *TRuntimeFlow* is created, and multiple instances of *TActionlet* are inserted using the method `InsertActionlet`. All *TActionlet* instances form a double linked list as an actionlet chain. The status of an actionlet can be characterized by three factors: 1) whether it has spaces to accumulate new data; 2) whether it has accumulated data that it can process right now; and 3) whether it has processed data to output. A workflow is ‘resumable’ if it can do something to move the intercepted traffic through the actionlet chain. The ‘resumable’ status depends on the status of all actionlets in the chain. The `Resumable` method in *TRuntimeFlow* invokes `GetStatus` for all *TActionlet*(s) in the chain to see whether 1) there is any actionlet that has data ready to process, or 2) there is any pair of adjacent actionlets where the first one has processed data and the next one has spaces. To resume a workflow, it reads data from the intercepted connection, feeds data to the entry actionlet *TActionletEntry* if it has spaces, invokes each actionlet to process its accumulated data, and feeds data from one actionlet’s output point to the next one’s input point through a pair of methods – `OutputData` and `InputData`.

The implementation of any derived *TActionlet* class must be self-contained. An actionlet should not rely on other actionlets for its own functionality. For instance, an actionlet should not assume that actionlets in front of it in the chain can buffer data for it. If the actionlet needs to hold a complete data object before processing, it must prepare enough buffer spaces to store the object. Otherwise, it might stall the processing when it is waiting for data but there is no buffer space to read from the intercepted connection.

Block Data Descriptor:	
NestLevel	: <i>Current nest level of paired actionlets</i>
ContentType[NestLevel]	: <i>1 (for meta-infomration) 0 (for data)</i>
Index[NestLevel]	: <i>Index of block</i>
Name[NestLevel]	: <i>What the block is</i>
BodyIndex[NestLevel]	: <i>Index for multiple blocks of the same data object</i>
Body	: <i>Pointer to and size of data content for the block</i>

Figure 19: Descriptor for Block Exchanged between Actionlets

4.3.3 Data Exchange among Actionlets

A workflow consists of actionlets for application layer protocols and for data objects wrapped in protocols. Protocol actionlets are paired for unwrapping and wrapping protocol information to ensure the integrity of interception. Because there might be multiple layers of protocols, we see a workflow as being comprised of nested protocol actionlet pairs with data manipulation actionlets in the middle, as illustrated in Figure 17. Suppose we need to apply access control on an email, where the actual data to be manipulated are the text body and the file attachments. For such manipulation, one actionlet is used to disassemble the email and another one to reassemble it. An email is disassembled into a body part and multiple attachment parts. Each part contains a header and data. The data manipulation actionlet is only interested in the data. Headers are useful for later actionlets for message reassembling.

Actionlets exchange data in blocks. The input blocks to an actionlet are categorized into three types: 1) blocks that contain data for the actionlet to process; 2) blocks that contain meta-information to help the actionlet; and 3) blocks which the actionlet does not understand. For an email unwrapping actionlet, a type 1 block can be partial data of an email message; a type 2 block can be an indicator of the end of transfer of an email message; a type 3 block can be a POP3 or SMPT command processed by the previous actionlet. An actionlet processes type 1 blocks, consumes type 2 blocks, and passes type 3 blocks to the next actionlet directly. Type 3 blocks are usually protocol related information output from

a protocol unwrapping actionlet and consumed by the paired wrapping actionlet at the end. For instance, a block of email headers are not understandable by the actionlet that processes the email attachment. But the block is used by the email wrapping actionlet to reconstruct the email.

Figure 19 shows the basic descriptor of an exchanging block. Because the actionlet chain consists of nested pairs of protocol unwrapping and wrapping actionlets, the block contains a nest level indicator and a stack for block specification. An protocol unwrapping actionlet increases the nest level and push the block specification into the stack. The corresponding wrapping actionlet pops the block specification from the stack and decrease the nest level.

4.3.4 Data Manipulation Actionlet

Sitting in the middle of the interception workflow, a data manipulation actionlet performs three key functionalities for the TBAC enforcement of TDPs. First, it identifies what kind of data objects are in communication. Second, it evaluates the surrounding context of the data recipient to get the current trust vector. Third, it acquires necessary restriction filters and applies them on data objects.

The meta-information about data objects, e.g. information related to the TBAC policy specification comes in three ways. It can be embedded inside of application layer protocols such as HTTP header fields or email MIME headers. A protocol unwrapping actionlet parses the information and relays it to a data manipulation actionlet. The meta-information can also be from a data object itself such as security related keywords etc. An actionlet can also digest the data object and query a session tracking agent with the digest to see whether the agent knows the object.

In our prototype, we use tags to represent the meta-information of protected data objects. Once a data manipulation actionlet gets an object's tag, it locates the right context vector agent through a TBAC engine and sends a query for the current trust vector. The

actionlet then uses the trust vector to determine the applied restriction filter names based on the TBAC specification.

All restrictions filters are implemented as exported functions in dynamically loadable libraries, e.g. shared objects in our prototype. An actionlet loads libraries based on restriction filter names, then acquires the filter functions, and invokes functions with the accumulated data objects to derive the suitably restricted objects.

4.4 Experiments

We evaluate the performance of interception mechanisms on a Intel Core 2 Duo 2.66 GHz machine with 4G memory on a Gigabit LAN except for certain measurements for the ProtectIT abstract. The system runs Xen virtual machine monitor 3.1. We perform measurement for guest virtual machines that run Linux with 1G memory.

We first show the micro-benchmark test of interceptions using the ProtectIT abstract. Then we switch to the proxy-based interception runtime. We measure the latency and achievable throughput on intercepted connections, and the CPU utilization too.

4.4.1 ProtectIT Micro-benchmark

The ProtectIT abstract is developed on Xen virtual machine monitor 3.0. The micro-benchmark is based on a Pentium IV 2.8GHz machine with Gigabit Ethernet links and 512M memory with Guest Linux systems using 128M memory each. We run micro-benchmarks to show where the overhead of ProtectIT comes from.

The first measurement shows the basic overhead posed by the interception service on a TCP/IP connection using ProtectIT. A small socket client application runs inside the guest virtual machine, receiving or sending a data set from/to a remote server. The data sizes range from 64 kilobytes to 1024 kilobytes. We compare five different cases to identify interception overheads. Figure 20 shows the time consumed to receive one unit of the data set for all cases. The first column is the baseline measurement for connections without ProtectIT enabled. The second column is the case when the connection does not match any

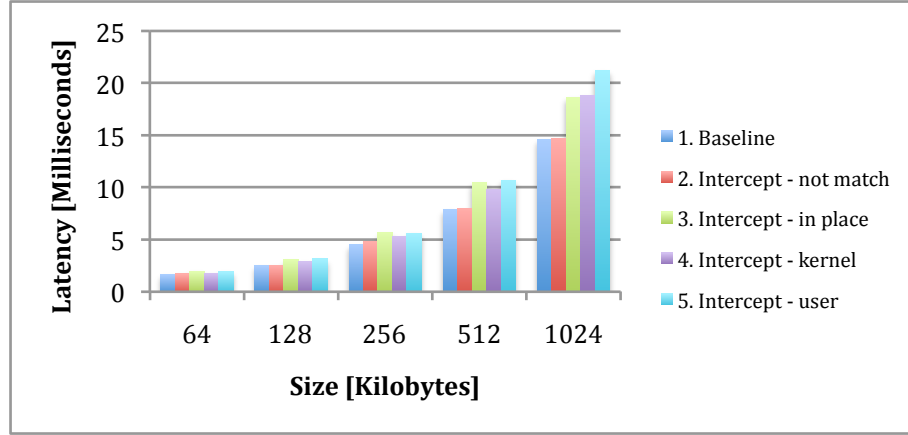


Figure 20: Overhead of Intercepted Inward Traffic using ProtectIT

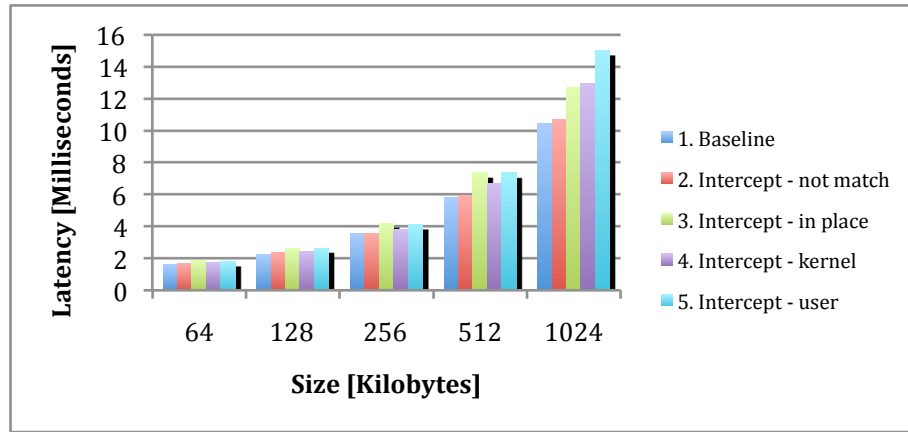


Figure 21: Overhead of Intercepted Outward Traffic using ProtectIT

rule. The third to fifth columns are all intercepted connections with different operations. The third case directly processes the packet on the network receiving or sending path. The fourth case buffers the data and then uses a kernel level dummy-copy actionlet. The fifth case uses a user level runtime to run a dummy-copy actionlet.

The first two columns show almost the same results, which indicates that the overheads for hooking interception services are negligible. The third shows that reconstructing new packets, along the critical path of packet processing induces moderate overheads. That is why it performs worse than the kernel level actionlet case of the fourth column. Application

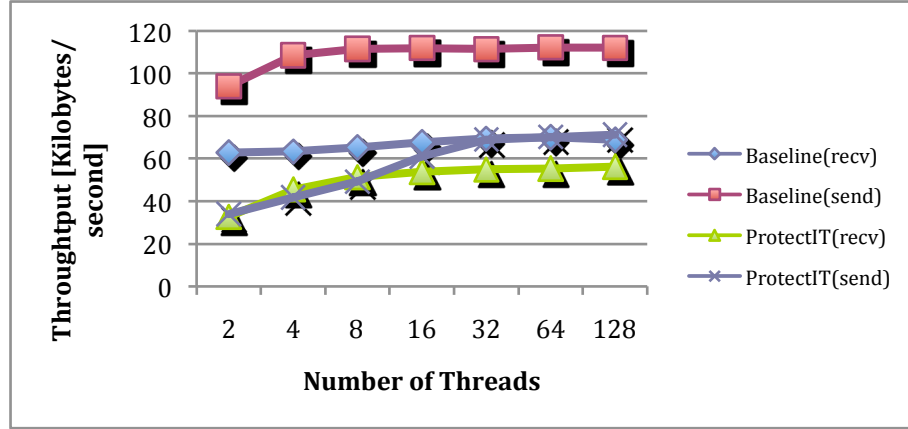


Figure 22: Throughput of Intercepted Traffic using ProtectIT

level actionlets incur more overhead than those kernel level ones due to scheduling overhead and additional copying and user-kernel boundary crossing. We observe very similar results from figure 21 for intercepting outward data.

In figure 22, we use a multi-threaded client sending requests to a remote multi-threaded server. It shows throughput with different number of threads. The results are similar for different request data sizes in the medium range, which is why we depict a mid range size of 256 kilobytes. For both incoming and outgoing traffic, we observe that the way we handle packet bursts has a negative impact on the Xen virtualized network, particularly affecting the exchange of packets between network front-ends and back-ends. As a result, the throughput achieved when doing interception is significant lower. Using different, unstable implementation can reduce this gap, which indicates that this difference is not inherent to the interception mechanism, but rather a consequence of our current inefficient implementation.

4.4.2 Proxy-based Interception Micro-benchmark

Our simulated network stack is a simple prototype to demonstrate the ProtectIT mechanism. Its implementation is inefficient when taking the achievable throughput into account and

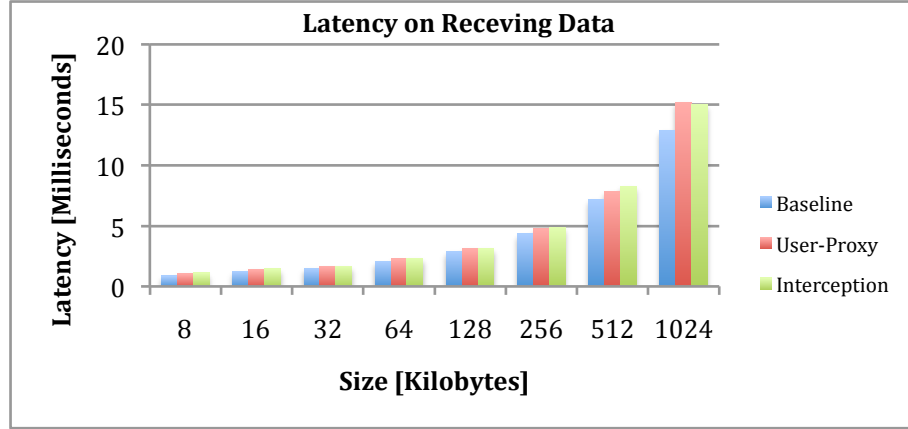


Figure 23: Comparison of Latency on Receiving Data

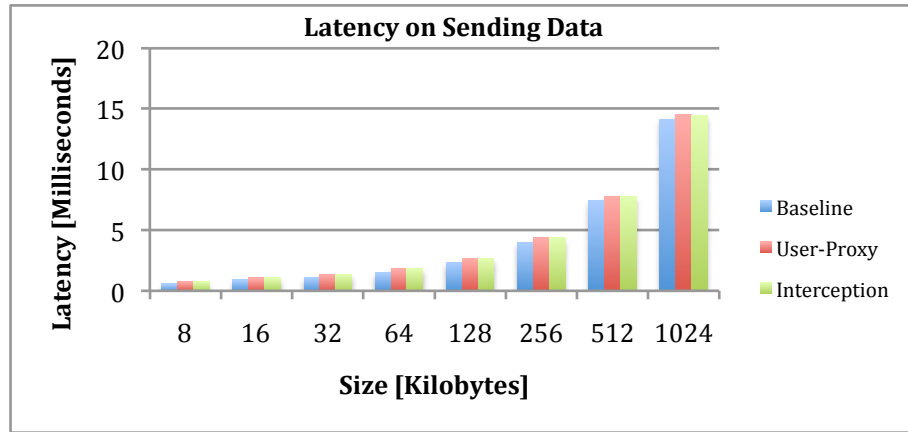


Figure 24: Comparison of Latency on Receiving Data

when there are out-of-order packets or dropped packets. On the other hand, the proxy-based interception uses the Dom0's network system which is stable and highly optimized. Here we measure the performance of the proxy-based interception in the aspect regarding latency, throughput and CPU utilization.

We evaluate the costs of the data protection enforcement mechanism in an interception runtime. Figure 23 shows experimental results for an intercepted client-server connection. A guest client sends a request to a remote server, and then receives N bytes of reply data.

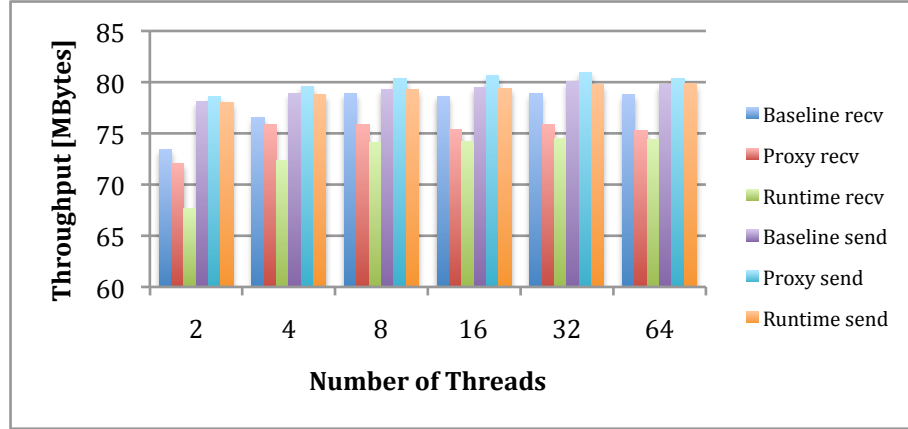


Figure 25: Throughput Comparison

The columns from left to right in the figure show the latency for the cases of (1) the baseline client-server connection, (2) using a client-aware proxy, and (3) using a transparent interception runtime. The figure shows that our interception runtimes perform almost identically to the client-aware proxy. This is because packets go through exactly same paths for these two cases. There are only small amounts of work being performed for manipulation of TCP/IP addresses and adjusting checksums. For medium to large data size, proxy-based solution incurs extra delay on latency for around 10-15%.

Figure 24 shows similar results for sending out N bytes of data. The gap between the baseline connection and the proxy-based connection is much smaller here. This is because of the high available bandwidth between the guest domain and the Dom0 when moving outgoing data from the guest to the dom0.

The throughput achievable when using TDPs is measured in Figure 25. Here, a multi-threaded client sends requests for downloading or uploading. Our interception runtime lags behind baseline connections by only about 6%, and it lags behind client-aware proxies by about 2%. For uploading data, it performs almost the same as the baseline, and is only slightly slower than the client-aware proxy. Considering that large volume traffic is usually observed at servers, this implies that throughput matters more for data being sent out, for which case our interception solution is entirely acceptable.

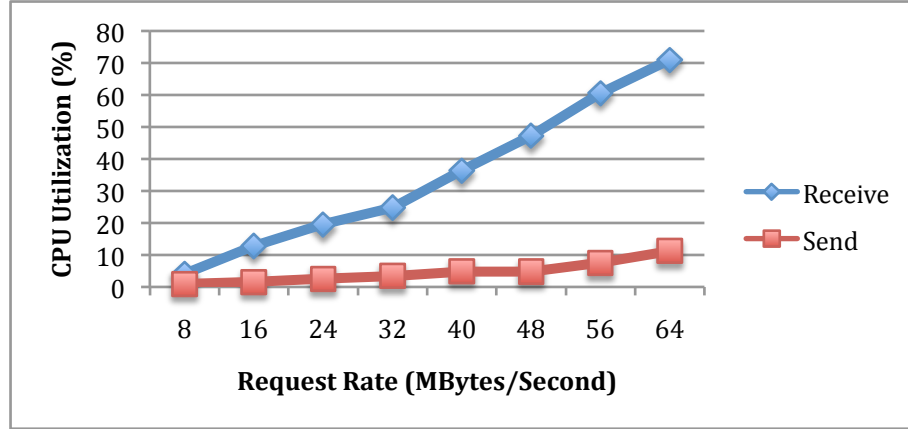


Figure 26: CPU Utilization of Proxy-based Interception

We then measure the CPU utilization for proxy-based interception in figure 26. Here, we use 16 client threads sending requests for downloading or uploading data of 1 megabytes. As shown in the figure, processing outgoing traffic (uploading) by the interception runtime incurs only moderate CPU overheads. It is because the guest VM can write data very quickly to the Dom0 due to high available bandwidth between the guest domain and the Dom0. The interception runtime can then read in large chunk of data each time. On the other hand, processing incoming traffic (download) incurs large CPU overhead, it is because that network packets tend to arrive in small blocks. The interception runtime tries to read data as soon as it comes, so it frequently check the socket and moving data from the kernel space to user space. Comparing the overhead of two cases, we believe that the high overhead for incoming traffic is not because that the interception runtime needs so much processing power, but rather implementation related.

We next measure overheads of TDPs on a typical end user application. Figure 27 shows that a HTTP response is intercepted. The HTTP message includes a header that contains our additional meta-information fields, and a content body that needs to be protected. We show in the figure the latency of (1) the baseline of an unprotected connection, (2) using the transparent proxy and processing the data as it arrives, and (3) using the transparent proxy and processing data only after receiving the complete data content. The increased latency

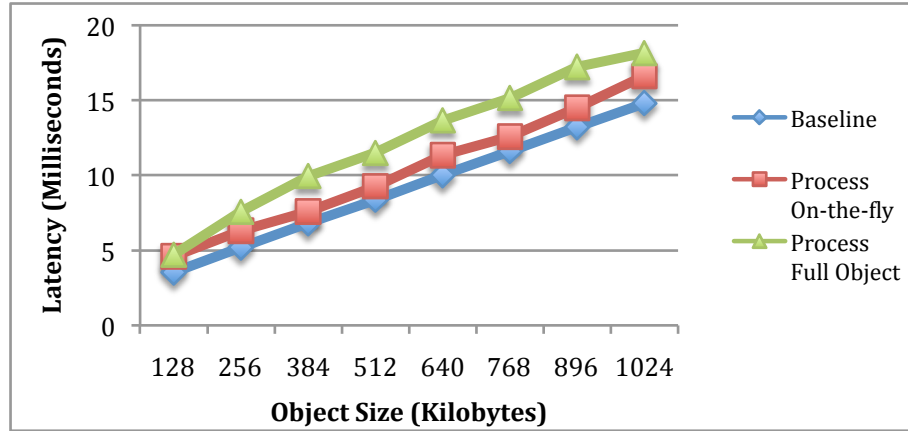


Figure 27: Latency on HTTP Traffic

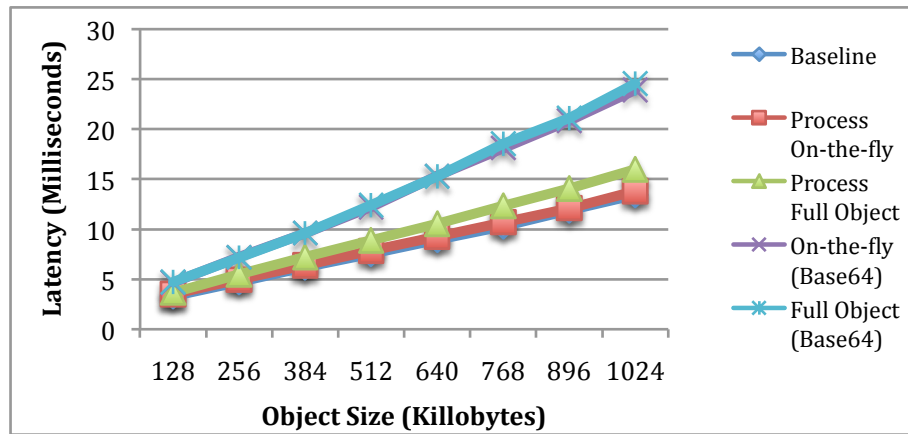


Figure 28: Latency on SMTP Traffic

when using a transparent proxy is about 10% if it is able to process partial data as it arrives. If it needs to hold the data in some buffer, the delay increases to 20-30% for medium to large data sizes.

We next measure the overheads of TDPs on an email client application. Figure 28 shows that an intercepted SMTP session. Email messages with attachments of different sizes are sent out. Attachments are in plaintext mode or base64 encoded. We show in the figure the latency of (1) the baseline of an unprotected connection, (2) using the transparent proxy and processing the attachment in plaintext as it arrives, and (3) using the transparent

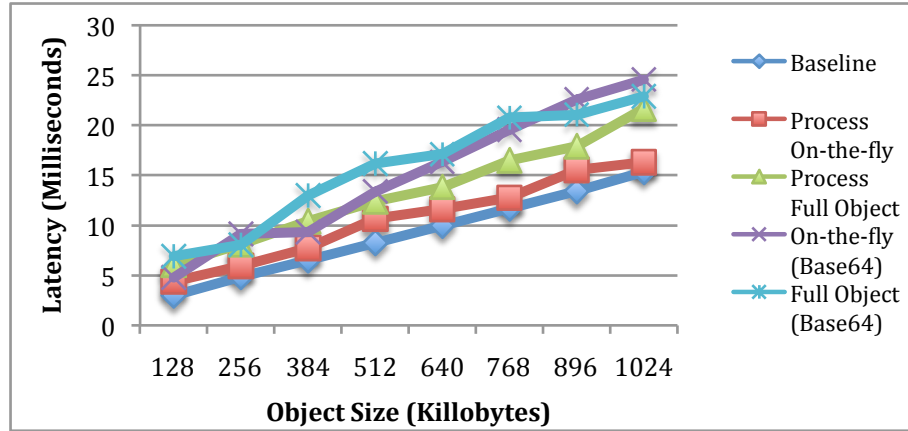


Figure 29: Latency on POP3 Traffic

proxy and processing data after receiving the complete plaintext attachment. Case 4 and 5 are same as 2 and 3 except that attachments are now base64 encoded. As shown in the figure, processing the plaintext attachment as it arrives incurs only slight delay than the baseline case. This is consistent with the micro-benchmark test in figure 24. Waiting for the full attachment incurs moderate delay of around 20%, and processing base64 encoded attachment increase the delay to 80%.

Figure 29 shows an intercepted POP3 session with same experiment cases. It shows similar results as the intercepted SMTP session. The standard deviation of measured latency is large so the figure is not as smooth as the SMTP case. We observe that even a small time change when reading in a packet might change final latency results significantly.

CHAPTER V

TRUSTED DATA PATH

The context flow control (CFC) approach utilizes a trust-based access control (TBAC) model to protect data against inappropriate information disclosures due to potentially compromised systems and/or faulty or malicious applications. It deploys a TBAC policy on all nodes along the data delivery path to establish a trusted data path (TDP) across providers, intermediate nodes, and end nodes. In this chapter, we first describe the TDP software components for TBAC policy specification, for TDP recognition, and for TBAC enforcement. We next explain how to deploy those components into an existing distributed system. We then present application examples to demonstrate the usefulness of the TDP software.

5.1 TDP Software Components

5.1.1 Access Restrictions and Trust Vectors

Access restrictions and trust vectors are two basic elements of the TBAC policy specification. Access restrictions define permissions on protected data objects. Trust vector formulas define the mapping from the active context to the quantification of risks of inappropriate information disclosures. Combined with a mapping from trust vectors to access restrictions, we have the specification of a TBAC policy.

TDP software defines access restrictions as filter functions. The filter functions are organized based on restriction names and types of data objects (e.g. tags) on which filters apply. All filters for the same object type are grouped together into one dynamically loadable library. A library is defined as:

$$Library := (\text{shared object, data object tags, } \{ (\text{restriction name, restriction filter}) \})$$

TDP software supports queries for libraries based on data object tags, and queries for restriction filters based on restriction names per library.

The trust vector formula can be either implemented as a standalone context agent or as a loadable library which can then be integrated into the interception runtime dynamically.

TDP software maintains a database for formulas as following:

$Table_{data2formula}$	$:=$	$\{ (tags, formula) \}$
$Table_{formula2context}$	$:=$	$\{ (formula, context\ agent) \}$
$Table_{formula2library}$	$:=$	$\{ (formula, shared\ object) \}$
$Table_{formula2restriction}$	$:=$	$\{ (formula, result, restriction\ name) \}$

It specifies the association of data types (tags) and formulas, context agents on which a formula depends, libraries or agents in which formulas are implemented, and most importantly how to map calculated trust vectors to names of access restrictions.

5.1.2 Interception Rules and Runtimes

The CFC approach focuses on network connections, particularly TCP/IP connections between communicating parties. It traces all connections that carry protected data objects, and put them under control of TBAC engines to form TDPs. TDP software specifies interception rules that define 1) the above TCP/IP connections by IP addresses and ports; 2) the interception runtimes that are capable of processing these connections; and 3) the nodes on which to intercept. An interception rule is defined formally as following:

$Pattern_{IP}$	$:=$	(network address, subnet mask)
$Pattern_{port}$	$:=$	(minimum, maximum)
$EndPoint$	$:=$	($Pattern_{IP}, Pattern_{port}$)
$Location$	$:=$	node identifier
$RuntimeName$	$:=$	interception runtime name
$InterceptionRule$	$:=$	($EndPoint, EndPoint, Location, RuntimeName$)

Interception runtimes are executable programs that process application layer protocols and protected data objects in the TDP software. They are identified by distinct names that describe their abilities. For instance, a ‘Care2X-HTTP-HTML’ runtime handles HTTP

connections carrying HTML documents between client browsers and the front-end Web server in the Care2X healthcare system.

The conversion from normal data delivery paths to trusted data paths proceeds in the following order. Interception rules are translated into hooks in Dom0(s) of the specified nodes, and related interception runtimes are up and running. Intercepted traffic is then redirected to proper interception runtimes. Inside an interception runtime, protocol-related actionlets process application layer protocols and parse any meta-information (tags) about the contained data objects. Tags are then mapped to restriction libraries and trust vector formulas. The data manipulation actionlets invoke the desired formulas, get the trust vector, and locate the restriction filters. The interception runtimes then apply the restriction filters and push the processed data objects back with rewrapped application layer protocols. As a result, suitable restricted versions of the protected data objects are delivered to recipients.

5.2 *Deploy TDP Software*

Trusted data paths cannot be created without the existence of trusted places that are safe for performing monitoring and enforcing trust-based access controls. Our prototype uses the control domains of the Xen para-virtualized platform as such trusted places, and the implementation assumes the use of Xen throughout the TDP framework. The implementation also uses one centralized management server that maintains all the software components needed and deploys them to participants. We have not yet considered scalability or reliability issues that would require further enhancements of this simple implementation.

5.2.1 *Deployment Interface*

By default, we treat the management server as the root-of-trust. For each node that qualifies for TBAC deployment, there is a trusted data path frontend (*TDPFront*) running in the Dom0 of the node. The *TDPFront* establishes a mutually authenticated secure connection with the management server. In this way, the management server obtains a list of all qualified nodes, and can certify the existence of a TBAC engine and the trustiness of context

agents in the node's Dom0.

The *TDPFront* presents an interface for the management server to 1) query the operating system information of the Dom0; 2) deploy executable files, dynamically loadable libraries, and kernel modules to desired locations; and 3) run the executable files or load the modules. A file deployment message from the server deliver a named file to the *TDPFront* who then stores it in the specified directory and invokes the embedded command. The message contains elements as illustrated blow:

Type	: application, or shared object, or kernel module
Descriptor	: unique identifier linking to attached file
Version	: version of attached file
Directory	: location to store attached file
Name	: name of attached file
Command	: execute attached file or load it as module
Data	: of attached file

Files with newer versions can replace existing ones. The data element is optional when the files are already deployed. The server also supports the poll mode which allows the *TDPFront* to issue a deployment message reversely asking for certain components. The server uses the type and descriptor information to locate the files and issues new messages to deliver the requested files.

Other than the *TDPFront*, there are other essential components for TDPs on each of the participated nodes such as *iccore* for driver level hooks, *nettcp* for network stack simulators, *pathctrl* for interception manager modules, *netproxy* for proxy manager modules, and *tevent* for the event middleware etc. All those components need to be in place before the node can join any trusted data path.

Components for setting up a trusted data path include interception rules, associated interception runtimes, the restriction filter libraries, and the trust vector formulas. The management server maintains the whole set of interception rules. Every time a node boots up, the *TDPFront* is configured to run automatically. It contacts the server by establishing a secure connection. The server enumerates all interception rules matching the node, and

sends rule-setup messages. The *TDPFront* responds the messages by setting the rules up in either *nettcp* or *netproxy*. At the same time, the server deploys and invokes the interception runtimes specified in the rules. If the interception rules have associated components like restriction libraries, trust vector formulas, and depended context agents, the server deploys them too. Otherwise, when actual interception happens, the interception runtime will request for missing pieces through the *TDPFront* using reverse deployment messages to the management server.

5.3 Applications

5.3.1 Care2X — Healthcare Information System

In the Care2X system, the backend database is the source of protected patient information. The Web server works as a frontend to grant to end users permissions to access certain patient records. The data delivery path therefore, involves the backend database node, the Web server node, and various end user nodes. While core system nodes like the database and web server are typically equipped with sufficient security measures, such assumptions cannot be made about end-user nodes that may be configured in various ways and operate in changing environments, thus presenting different risks of information misuse. Evaluating the potential risks and determining appropriate access restrictions depends on proper interpretations of the security and privacy policies in place. We demonstrate the usefulness of the TDP abstraction for the Care2X application using the following settings.

Protected patient records in Care2X contain both personal identifiers and treatment records. So, a HTML page served to end users is tagged as *Care2X* for tracking policies, *HTML* for message format, and *identifier* and *treatment* for its content. For fields belonging to the *identifier* group, we define three restriction filters that separately blank out critical personal information such as social security number (SSN), or general data such as names and addresses, or hospitalization related items such as patient-id. For fields belonging to the *treatment* group, we define two restrictions that separately blank out important data like

diagnosis and doctor notes, or routine checks like body temperatures. All filters are implemented in two libraries associated with tags (*Care2X*, *HTML*, *identifier*) and (*Care2X*, *HTML*, *treatment*). Individual filters are assigned unique restriction names.

We consider risks of information misuse with respect to two items of contextual information. One is the reasonableness of the access operations. The other is the security of the client side system. We classify reasonableness as $\{high, medium, low, not\}$, and client system security as $\{strong, weak, unknown\}$. We then associate these evaluations with different restriction names, as described earlier in tables 1 and 2. The above classification of reasonableness and security depends on formulas translated from security and privacy policies. In our prototype, we evaluate reasonableness based on access locations, time, etc, and we evaluate system security based on the detection of suspicious processes, listening ports, and the existence of unexpected network connections.

An interception rule is applied as a response to all of the risks listed above. It is deployed to all TBAC-enabled nodes, where each such rule specifies the Web server by its IP address and listening port. The rule runs with a linked runtime that is able to perform HTTP processing (i.e., it can handle the application layer protocol) and can apply the restrictions mentioned above. For machines that are not capable of TBAC enforcement, we can deploy a different interception rule at the server side that applies restrictions based on whether the client machine is TBAC-enabled.

5.3.2 Online-storage and Email

Another sample application – cooperative data sharing – is used to demonstrate the propagation control ability of trusted data paths. In this application, the data delivery path can be from client to online storage, from client to client via mail servers, or a composition of both. In addition to the risks considered in the earlier example, we also evaluate the possibility of improper data propagation. In other words, we ask the following question: when recipients of protected information try to relay such information, are the related protection

policies still in effect?

Propagation control depends on who has access to the protected information at the recipient side. If the recipient stores the data into secure places or uses protection policies when relaying data, we can say the information is still under control. Of course, applications run by participating parties may claim such controls but then not use them. In response, the TDP implementation uses a context agent to observe whether some connection belongs to such an application, and further, whether the application actually does what it claims. This again demonstrates the best-effort nature of TDP, where the observations made depend on the applications in use and the security concerns of participating entities.

To demonstrate the utility of TDP, we have developed a specialized client able to access Web server-based online storage and send emails. It claims that it will neither save data to files nor send out non-tagged email attachments. If the email server is part of a TDP, then (1) tagged emails remain under its control, (2) its propagation control abilities may be described as ‘whether the connection is from such a client’ and ‘whether it saves files’, and (3) we can verify whether outgoing emails are properly tagged. Toward that end, we compare attachments with the files opened by this application and with the data it acquires from online storage. This is done via a session tracking agent that keeps records of the tags and information digests of protected information items. This permits us to evaluate whether attachments match any protected information. Comparisons based on information digests made for this purpose rely in their correctness on the rarity of collision of such hash functions (SHA1 in our prototype) [12].

5.3.3 Demonstration

We have conducted experiments for above two applications to demonstrate the usefulness of the TDP software. Figure 30(a) shows part of a patient information page delivered for a highly reasonable access and 30(b) for a medially reasonable access. The social security number is removed from the page in the second case. The context agent used here queries

Cellphone. 1	4040000000
Email	jane.doe@care2x.org
SSS Nr.	901010001
Registered by	DoctorA

(a) Patient Information (High)

Cellphone. 1	4040000000
Email	jane.doe@care2x.org
SSS Nr.	
Registered by	DoctorA

(b) Patient Information (Medium)

Figure 30: Patient Information Shown on Different Reasonable Levels

for the serial numbers of installed devices to determine the access location, and the trust vector formula calculates reasonableness using the location information.

Care2X generates above HTML pages using pre-defined templates. Because of the variety of dynamically generated pages, it is impractical to write a filter that can process all pages directly. Instead, we examine the templates that Care2X uses, and mark the proper data fields with meta-information tags. The generated HTML pages will then include these tags. Actionlets in the interception runtime can locate these tags for policy related meta-information, and filters can parse the HTML pages to find the right places to apply restrictions. For example, the above pages use a template contain one line of:

{{ \$sSSSNr }}

representing the social security number in our experiment. We add tags to that line so it changes to:



<TDP reasonable="high"> {{ \$sSSSNr }} </TDP>

The generated page will contain HTML code as:

```
<TDP reasonable="high"> <tr>
  <td class = "reg_item"> SSS Nr. </td>
  <td class = "reg_item"> 901010001 </td>
</tr> </TDP>
```

Date	Doctor's daily notes	Details	By	Encounter Nr.
24/09/2009			DoctorA	2009000000

(a) Doctor's Daily Notes

Date	Diet plan	Details	By	Encounter Nr.
25/09/2009	Diet Plan: Breakfast - Diet A2; Lunch - Diet B2; Dinner - Diet C2; [Diet A2;B2;C2]		DoctorA	2009000000
24/09/2009	Diet Plan: Breakfast - Diet A1; Lunch - Diet B1; Dinner - Diet C1; [Diet A1; B1; C1]		DoctorA	2009000000

(b) Notes of Diet Plan

Figure 31: Treatment Related Notes Shown on Medially Secured System

The next example in figure 31 shows that different treatment related notes have different requirements on environment. We here determine the trust vectors based on whether there are suspicious connections (to known backdoor ports). In a medially secure context, figure 31(a) has the doctor's daily notes removed, and figure 31(b) keeps the diet plan notes in place. Care2X uses the same template to generate HTML pages for different notes, so we cannot directly tag a field with its security requirements. Instead, we tag the category filed, e.g. the one showing "Doctor's daily notes" or "Diet plan", and then link the data fields to the category tags.

Next we demonstrate the restricted accesses on images from an Web server. Because an image can hardly contain the TDP meta-information by itself, we assume that the server attaches meta-information as extra HTTP header fields. We show two different filters in figure 32. The watermarking filter is useful for protecting one's property, and the face blocking filter [31] is useful for protecting one's privacy. The context for determining the trust vectors in this case include whether the client side program is audited, and whether it makes connections to IP addresses not authorized. We also implement a PDF text extractor filter to demonstrate that we can provide partial access to a PDF document by extracting select portion of the document such as the 'Abstract'.

Table 8: Overhead of Various Filters on an Intel C2D 2.6GHz Node

Filter Type	Time (ms)
Remove tagged data in a 18 kilobyte HTML	0.081
Extract text from a 12 page PDF	93.18
Add watermark on a 640x428 JPEG	22.67
Blocking human faces on a 640x428 JPEG	353.48

As for propagation control, we automatically apply the watermarking filter if the client application is not the audited one. We also monitor its outgoing connections to SMTP servers. If the email recipients' addresses are from different domains then the owner of the data, we apply the face blocking filter on images, and the PDF abstract filter on PDF documents.

Table 8 lists four filters we use above — a HTML filter that removes certain tagged fields, a PDF text extractor that extracts abstracts from papers, an image watermarking filter, and a human face blocking filter. As shown in the table, filtering costs range from trivial for the HTML filter, to moderate for the watermarking filter, to fairly expensive for the PDF text extractor and the face blocking filters. The TDP's innate monitoring overheads are small, that its interception overheads are moderate as shown in previous chapters. The filtering overheads when applying restriction filters entirely depend on the applications in use and the filters they desire.



(a) Original JPEG Image



(b) JPEG Image with Watermark



(c) JPEG Image with Face Blocked

Figure 32: Restriction Filters on JPEG Images

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This thesis addresses data sharing between services and applications that are suspected of faulty or malicious behaviors or that run on compromised systems. Toward these ends, it develops and presents a context flow control (CFC) model used to implement with *trusted data paths* (TDPs). TDP software uses context-monitoring agents to continually analyze system and application service behaviors to derive their current levels of trust. It ensures that data flowing to target users will match their current trust levels using transparent traffic interception mechanisms, specified by trust-based access control (TBAC) policies. It also ensures end-to-end propagation controls on data, by enforcing TBAC along entire data delivery paths.

There remain multiple issues with the current TDP implementation. This includes its use of configuration files and associated static configuration methods. While these are well-suited for longer running data center applications and relatively static multi-tier web services, they are not likely appropriate for next generation dynamic service based systems in which services freely and dynamically form associations and service agreements with each other. Also statically configured is TDP's knowledge about the trusted places (i.e., Xen's Dom0 in our prototype) where TBAC engines can be deployed. Discovery protocols and directory support are needed to remove restrictions like these. Another issue is the potential fragility of the 'chain of trust' in the current Xen implementation, due to the complex nature of the software running in Dom0. This can be addressed with hardware support like the trusted platform module.

REFERENCES

- [1] ABERER, K. and DESPOTOVIC, Z., “Managing trust in a peer-2-peer information system,” in *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, (Atlanta, Georgia, USA), pp. 310–317, 2001.
- [2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARFIELD, A., “Xen and the art of virtualization,” in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, (Bolton Landing, NY, USA), pp. 164–177, 2003.
- [3] BELL, D. E. and PADULA, L. L., “Secure computer systems: Unified exposition and multics interpretation,” Tech. Rep. MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, 1976.
- [4] BERTINO, E., BONATTI, P. A., and FERRARI, E., “TRBAC: A temporal role-based access control model,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 191–233, 2001.
- [5] BIBA, K. J., “Integrity considerations for secure computer systems,” Tech. Rep. MTR-3153, Rev. 1, MITRE Corp., Bedford, MA, 1976.
- [6] CARE2X, “<http://www.care2x.org/>.”
- [7] CHESWICK, W., BELLOVIN, S., and RUBIN, A., *Firewalls and Internet security*. Addison-Wesley professional computing series, Boston, MA: Addison-Wesley, 2 ed., 2003.
- [8] COVINGTON, M. J., LONG, W., SRINIVASAN, S., DEV, A. K., AHAMAD, M., and ABOWD, G. D., “Securing context-aware applications using environment roles,” in *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, (Chantilly, Virginia, United States), pp. 10–20, 2001.
- [9] DAMIANI, M. L., BERTINO, E., CATANIA, B., and PERLASCA, P., “GEO-RBAC: A spatially aware RBAC,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, no. 1, p. 2, 2007.
- [10] DENNING, D. E., “A lattice model of secure information flow,” *Communications of the ACM*, vol. 19(5), pp. 236–243, 1976.
- [11] DIERKS, T. and ALLEN, C., “The TLS protocol version 1.0.” <http://www.ietf.org/rfc/rfc2246.txt>, 1999.
- [12] EASTLAKE, D. E. and JONES, P. E., “US Secure Hash Algorithm 1 (SHA1).” <http://www.ietf.org/rfc/rfc3174.txt>, Sept. 2001.

- [13] EISENHAUER, G., BUSTAMANTE, F. E., and SCHWAN, K., "A middleware toolkit for client-initiated service specialization," in *Operating Systems Review*, pp. 18–20, 2000.
- [14] EUROPEAN COMMUNITIES, "Directive on the protection of personal data (9546/ec)," Oct 1995.
- [15] FERRAILOLO, D. and KUHN, R., "Role-based access control," in *In 15th NIST-NCSC National Computer Security Conference*, (Baltimore, MD), pp. 554–563, Oct 1992.
- [16] FERRAILOLO, D. F., SANDHU, R., GAVRILA, S., KUHN, D. R., and CHANDRAMOULI, R., "Proposed NIST standard for role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.
- [17] FOGLA, P. and LEE, W., "Evading network anomaly detection systems: formal reasoning and practical techniques," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, (Alexandria, Virginia, USA), pp. 59–68, 2006.
- [18] FOGLA, P., SHARIF, M., PERDISCI, R., KOLESNIKOV, O., and LEE, W., "Polymorphic blending attacks," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, (Vancouver, B.C., Canada), 2006.
- [19] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., and LONGSTAFF, T. A., "A sense of self for Unix processes," in *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), p. 120, 1996.
- [20] FREIER, A., KARLTON, P., and KOCHER, P., "The SSL protocol version 3.0." Internet Draft.
- [21] GARFINKEL, T. and ROSENBLUM, M., "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, (San Diego, CA), pp. 191–206, February 2003.
- [22] GIANG, P. D., HUNG, L. X., LEE, S., LEE, Y.-K., and LEE, H., "A flexible trust-based access control mechanism for security and privacy enhancement in ubiquitous systems," *Multimedia and Ubiquitous Engineering, International Conference on*, vol. 0, pp. 698–703, 2007.
- [23] GU, Y., McCALLUM, A., and TOWSLEY, D., "Detecting anomalies in network traffic using maximum entropy estimation," in *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, (Berkeley, CA, USA), pp. 32–32, 2005.
- [24] GUPTA, M., JUDGE, P., and AMMAR, M., "A reputation system for peer-to-peer networks," in *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, (Monterey, CA, USA), pp. 144–152, 2003.

- [25] JIANG, X., WANG, X., and XU, D., “Stealthy malware detection through VMM-based “out-of-the-box” semantic view reconstruction,” in *CCS ’07: Proceedings of the 14th ACM conference on Computer and communications security*, (Alexandria, Virginia, USA), pp. 128–138, Oct 2007.
- [26] JONES, S. T., ARPACI-DUSSEAU, A. C., and ARPACI-DUSSEAU, R. H., “Antfarm: tracking processes in a virtual machine environment,” in *ATEC ’06: Proceedings of the annual conference on USENIX ’06 Annual Technical Conference*, (Boston, MA), 2006.
- [27] JONES, S. T., ARPACI-DUSSEAU, A. C., and ARPACI-DUSSEAU, R. H., “VMM-based hidden process detection and identification,” in *VEE ’08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (Seattle, WA, USA), pp. 91–100, March 2008.
- [28] JOSHI, J., BERTINO, E., LATIF, U., and GHAFOR, A., “A generalized temporal role-based access control model,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, pp. 4–23, Jan. 2005.
- [29] JUNG, J., SHETH, A., GREENSTEIN, B., WETHERALL, D., MAGANIS, G., and KOHNO, T., “Privacy Oracle: a system for finding application leaks with black box differential testing,” in *CCS ’08: Proceedings of the 15th ACM conference on Computer and communications security*, (Alexandria, Virginia, USA), pp. 279–288, 2008.
- [30] KALYAN, C. and CHANDRASEKARAN, K., “Information leak detection in financial e-mails using mail pattern analysis under partial information,” in *AIC’07: Proceedings of the 7th Conference on 7th WSEAS International Conference on Applied Informatics and Communications*, (Vouliagmeni, Athens, Greece), pp. 104–109, 2007.
- [31] KIENZLE, W., BAKIR, G. H., FRANZ, M. O., and SCHÖLKOPF, B., “Face detection — efficient and rank deficient,” in *Advances in Neural Information Processing Systems 17* (SAUL, L. K., WEISS, Y., and BOTTOU, L., eds.), pp. 673–680, Cambridge, MA: MIT Press, 2005.
- [32] KNUTSSON, B., LU, H., MOGUL, J., and HOPKINS, B., “Architecture and performance of server-directed transcoding,” *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 4, pp. 392–424, 2003.
- [33] KONG, J., GANEV, I., SCHWAN, K., and WIDENER, P., “CameraCast: Flexible access to remote video sensors,” in *In Proceedings of the Fourteenth Annual Multimedia Computing and Networking Conference (MMCN’ 07)*, (San Jose, CA), Jan. 2007.
- [34] KONG, J., SCHWAN, K., LEE, M., and AHAMAD, M., “ProtectIT: trusted distributed services operating on sensitive data,” in *Eurosys ’08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, (Glasgow, Scotland, UK), pp. 137–147, 2008.
- [35] KONG, J., SCHWAN, K., and WIDENER, P., “Protected data paths: delivering sensitive data via untrusted proxies,” in *PST ’06: Proceedings of the 2006 International Conference on Privacy, Security and Trust*, (Markham, Ontario, Canada), Oct 2006.

- [36] KOURAI, K. and CHIBA, S., “HyperSpector: virtual distributed monitoring environments for secure intrusion detection,” in *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, (Chicago, IL, USA), pp. 197–207, 2005.
- [37] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., and MORRIS, R., “Information flow control for standard OS abstractions,” in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, (Stevenson, Washington, USA), pp. 321–334, 2007.
- [38] KVM, “Kernel-Based virtual machine.” <http://www.linux-kvm.org/>.
- [39] LINUX, “Secure-Enhanced Linux.” <http://www.linux.org/>.
- [40] MAHONEY, M. V., “Network traffic anomaly detection based on packet bytes,” in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, (Melbourne, Florida, USA), pp. 346–350, 2003.
- [41] MAHONEY, M. V. and CHAN, P. K., “Learning nonstationary models of normal network traffic for detecting novel attacks,” in *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, (Edmonton, Alberta, Canada), pp. 376–385, 2002.
- [42] MANSOUR, M. S., SCHWAN, K., and ABDELAZIZ, S., “I-Queue: Smart queues for service management,” in *In Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC'06)*, (Chicago, IL, USA), pp. 252–263, Dec. 2006.
- [43] MCAFEE, INC., “McAfee-antivirus software and intrusion prevention solutions.” <http://www.mcafee.com/>.
- [44] MICROSOFT INC., “Windows Hyper-V server.” <http://www.microsoft.com/hyper-v-server/>.
- [45] MYERS, A. C. and LISKOV, B., “A decentralized model for information flow control,” in *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, (Saint Malo, France), pp. 129–142, Oct 1997.
- [46] NORTON, INC., “Norton Internet Security.” <http://www.symantec.com/norton/internet-security>.
- [47] OLESON, V., EISENHAUR, G., PU, C., SCHWAN, K., PLALE, B., and AMIN, D., “Operational information systems: an example from the airline industry,” in *WIESS'00: Proceedings of the 1st conference on Industrial Experiences with Systems Software*, (San Diego, CA, USA), 2000.
- [48] OLSON, J. S., GRUDIN, J., and HORVITZ, E., “A study of preferences for sharing and privacy,” in *CHI 05: extended abstracts on Human factors in computing systems*, (Portland, OR, USA), pp. 1985–1988, 2005.

- [49] PAYNE, B. D., DE CARBONE, M. D. P., and LEE, W., "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, (Miami Beach, Florida, USA), pp. 385–397, 2007.
- [50] PAYNE, B. D., CARBONE, M., SHARIF, M., and LEE, W., "Lares: An architecture for secure active monitoring using virtualization," *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 233–247, 2008.
- [51] PETRONI, JR., N. L., FRASER, T., MOLINA, J., and ARBAUGH, W. A., "Copilot - a coprocessor-based kernel runtime integrity monitor," in *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, (San Diego, CA, USA), pp. 13–13, 2004.
- [52] PETRONI, JR., N. L., FRASER, T., WALTERS, A., and ARBAUGH, W. A., "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, (Vancouver, B.C., Canada), 2006.
- [53] PRIYANTHA, N. B., CHAKRABORTY, A., and BALAKRISHNAN, H., "The Cricket Location-Support System," in *6th ACM MOBICOM*, (Boston, MA), August 2000.
- [54] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., and YOUMAN, C. E., "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.
- [55] SESHASAYEE, B., NARASIMHAN, N., PAI, A. B. A., and SCHWAN, K., "VStore - efficiently storing v(irtualized) state across mobile devices," in *MobiVirt 2008, The Workshop on Virtualization in Mobile Computing*, (Breckenridge, Colorado, USA), June 17 2008.
- [56] SINGARAVELU, L. and PU, C., "Fine-grain, end-to-end security for web service compositions," in *2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA*, pp. 212–219, IEEE Computer Society, 2007.
- [57] SRIVASTAVA, A. and GIFFIN, J., "Tamper-resistant, application-aware blocking of malicious network connections," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, (Cambridge, MA, USA), pp. 39–58, 2008.
- [58] SZOR, P., *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.
- [59] TAKAGI, H. and ASAKAWA, C., "Transcoding proxy for nonvisual web access," in *Assets '00: Proceedings of the fourth international ACM conference on Assistive technologies*, (Arlington, Virginia, United States), pp. 164–171, 2000.
- [60] THOTTAN, M. and JI, C., "Anomaly detection in IP networks," *Signal Processing, IEEE Transactions on*, vol. 51, pp. 2191–2204, Aug. 2003.

- [61] TRAN, H., HITCHENS, M., VARADHARAJAN, V., and WATTERS, P., “A trust based access control framework for p2p file-sharing systems,” *Hawaii International Conference on System Sciences*, vol. 9, p. 302c, 2005.
- [62] UNITED STATES. GOVERNMENT ACCOUNTABILITY OFFICE., “Information security [electronic resource] : emerging cybersecurity issues threaten federal information systems : report to congressional requesters,” 2005.
- [63] U.S. CONGRESS, “The health insurance portability and accountability act,” 1996.
- [64] WANG, Y. and VASSILEVA, J., “Bayesian network trust model in peer-to-peer networks,” in *In Proceedings of Second International Workshop Peers and Peer-to-Peer Computing*, pp. 23–34, 2003.
- [65] WIDENER, P., SCHWAN, K., and BUSTAMANTE, F., “Differential data protection for dynamic distributed applications,” in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pp. 396–405, Dec. 2003.
- [66] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., and MAZIÈRES, D., “Making information flow explicit in HiStar,” in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, (Seattle, Washington, USA), pp. 263–278, 2006.
- [67] ZELDOVICH, N., BOYD-WICKIZER, S., and MAZIÈRES, D., “Securing distributed systems with information flow control,” in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, (San Francisco, CA, USA), pp. 293–308, 2008.